>>#<<. CLOS BROWSER

One of the key components of CLOS in inheritance. The CLOS Browser provides functionality for displaying this structure and for extending it. It also provides functions for displaying and changing the class definitions and method definitions which make up a system written in CLOS.

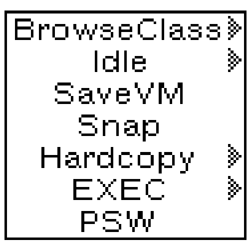
Creating a Browser

A browser can be createded in two ways:

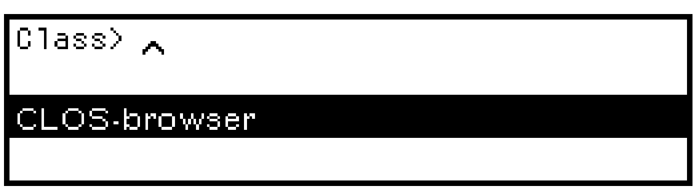
- Via a menu option from the Background Menu
- By calling the function CLOS-BROWSER:BROWSE-CLASS on a class

Creating a browser via the Background Menu

When the CLOS-BROWSER module is loaded, an enty is added to the Background Menu, as shown below:



Selecting the menu item BrowseClass brings up a window, with a prompt for the name of the class to use as the root of the browser as shown below.



Type in the name of the class you wish to browse at the flashing cursor, and the class graph will be drawn in the window.

Creating a browser programmatically

Browsers can also be created by calling the function BROWSE-CLASS:

(BROWSE-CLASS *&OPTIONAL CLASS-NAME-OR-LIST &KEY :WINDOW-OR-TITLE :GOOD-CLASSES :POSITION***)** **[Function]**

This function brings up a browser on the class named or the list of classes named. If a window is supplied for the *:WINDOW-OR-TITLE* argument, then the browser is created in that window, else an appropriately sized window is created. The window is positioned at the *:POSITION* argument or, if not supplied, then the position is set via the mouse. If a text string is supplied for the *:WINDOW-OR-TITLE* argument, then that string is used for the window title, else the string "CLOS-browse" is used. If *:GOOD-CLASSES* is supplied, then only those classes in the list are displayed.

Using the Class browser

Instances of CLOS-BROWSER are operated on through a mouse-based interface.

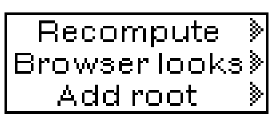
Buttoning on the browser will cause one of the following menus to be popped up:

- One menu appears when the left or middle button is pressed while the mouse is in the title bar. This menu has operations that apply to the browser itself.
- The other menu appears when the middle button is pressed when the mouse is on one of the nodes in the browser.

If the left button is pressed when the mouse is on a node, that node is boxed. This marks the node for some operations.

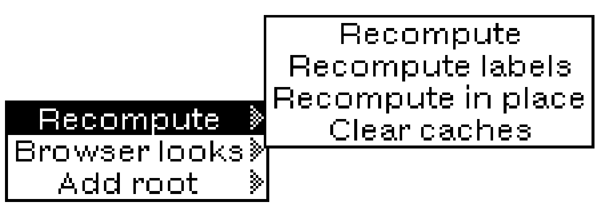
Options in the title bar menu

The following menu appears when you left- or middle-button in the title bar.



Recompute and it’s suboptions

Selecting the Recompute option and dragging the mouse to the right causes the following submenu to appear:



Most of these items change the appearance of the browser, not the contents.

Recompute
: Recomputes the browser from the starting objects. It does not recompute the labels for each node if those labels are cached in the Label-Cache slot of the browser.

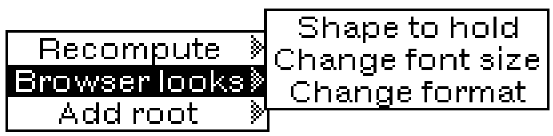
Recompute Labels
: Recompute the browser from the starting objects, including the labels.

Recompute inPlace
: Recompute the browser without affecting the scrolled location of the lattice within the window.

Clear caches
: Clear the caches of the nodes.

Browser looks and it's suboptions

Selecting the Browser looks menu item and sliding to the right causes the following submenu to appear:

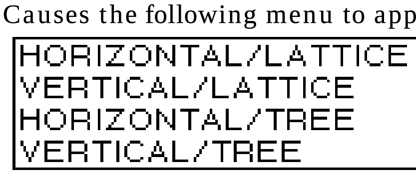


Selecting one of these options changes the looks of the browser.

Shape to hold
: Make the window for the browser just large enough to contain the browser.

Change font size
: Causes a menu of alternative font sizes to pop up. Selecting one of these causes the browser to be redrawn with the nodes at that font size.

Change format
: Causes the following menu to appear:



Horizontal/Lattice
: Lays out the grapher as an horizontal lattice.

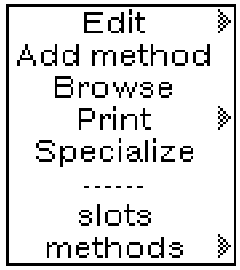
Vertical/Lattice
: Lays out the grapher as a vertical lattice.

Horizontal/Tree
: Lays out the grapher as a horizontal tree.

Vertical/Tree
: lays out the grapher as a vertical tree.

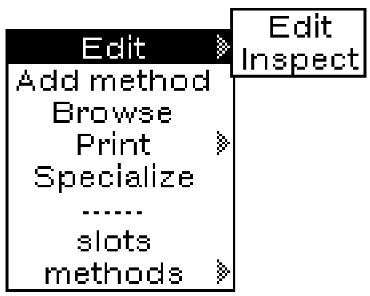
Options in the Middle-button menu

The following menu appears when you middle-button over a node in the graph:



Edit and it's suboptions

Selecting Edit causes an editor on the class definition to be brought up. Sliding the mouse to the right causes the following menu to appear:



Edit	Edits the class named by the node
Inspect	Inspects the class object named by the node.

Add Method

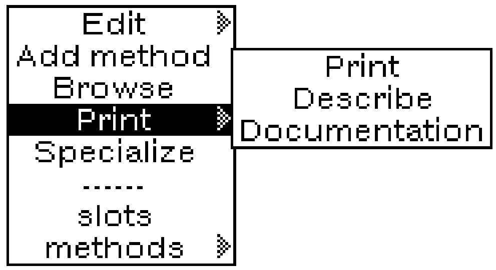
Selecting the Add Method option brings up an editor window with a template for a method to be added to that class. When the editor is done the method is installed for that class and the menu updated.

Browse

Selecting the Browse option causes a browser to be created starting with that class as the root.

Print and it's suboptions

Selecting Print prints out the class definition. Sliding the mouse to the right causes the following menu to appear:



Print	Print’s the class definition
Describe	Describes the class, listing it’s metaclass, it’s supers classes, it’s subclasses, it’s CPL, and the number of methods specialized to it.
Documentation	Print’s the documentation string for the class

Specialize

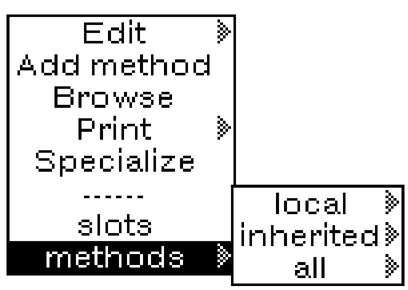
Selecting the Specialize option brings up an editor window with a template for a subclass to be added to that class. When the editor is done the class is installed and the browser updated.

Slots

Selecting the Slots option is the same as selecting the Edit option, it brings up an editor on the class definition.

Methods

The Methods option allows you to edit one of the methods defined for that class. Selecting it and sliding to the right brings up the following sub-menu:



Local	Bring up a menu of the local methods, ie methods directly defined for this class
Inherited	Bring up a menu of the methods this class inherits from it’s superclasses.
All	Bring up a menu of all the methods defined for this class, both local and inherited.

Selecting an item with the left button from the resulting menu brings up an editor on that method. If there are multiple methods that apply, a gray triangle appears in the right edge of the menu next to that item. Sliding to the right brings up a menu of method specializers to select the appropriate method.

Interlisp-D fixed allocations: conversion to Intermezzo

(all numbers in octal)

Name	Old Size (pages)	Old Addr	New Size	New Addr	Old Real	New Real
On page 26:						
Interface Page	1	26,10000	1	6,0	3	3
Primary Page Map	2	26,0	**10**	6,1000	2153	3313
StatsSpace (unused)	2	26,120000	**omit**			
Interrupt Table (unused)	?	26,121200	**omit**			
MiscStats	2 (1 used)	26,122000	2	6,5000		
UFN Table	2	26,123000	2	6,6000		
DTD ****	20	26,124000	20 (140)	6,10000		
MDS Type Table	40	26,100000	**1/2 seg**	6,100000	1600	2400
FPTOVP	1/4 seg	26,40000	**1 seg**	4,0	501	2000
Misc:						
Secondary Page Map	1/4 seg	25,0	**1 seg**	5,0		
Stack	1 seg	27,0	1 seg	1,0	1000	1400
GC Hash table	1/2 seg	73,0	1/2 seg	20,0	1400	2600
GC Collision *	1 seg	74,0	1 seg	21,0		
GC Overflow **	1	73,100000	1	20,100000	1640	3000
GC Big Ref	1-?	73,100400	1-?	20,100400		
Display Bitmap	312	76,0	312	22,0	1641	3001
LockedPageTable	—	(26,20000)	**20**	6,70000		
Map (Dlion only)	100	—	400	—	400	400
IOPage (Dlion only)	1	0,177400	1	0,177400	500	1000
SmallPosP's	1 seg	16,0	1 seg	16,0		
SmallNegs	1 seg	17,0	1 seg	17,0		
Arrayspace Start		40,0		23,0		
Atoms:			(if 64K atoms)			
Pname Pointers	1 seg	20,0	**2 seg**	10,0		
Definitions	1 seg	21,0	**2 seg**	12,0		
Topvals	1 seg	22,0	**2 seg**	14,0		
Property Lists	1 seg	23,0	**2 seg**	2,0		
Atom Hash Table	1/2 seg	24,0	**1 seg**	7,0		
Pname Chars ***	8 seg	30,0	**6 seg**	72,0		

* Collision table occupies 1 segment, all preallocated, for no particularly good reason. It wants to be big, because once it fills up, you have to disable gc. I have seen the table get as large as a quarter segment. Current algorithms prevent it from being larger than one segment, but it would be easy to make it 2 segments long.

** GC Overflow table is actually just a few words. Current microcode relies on it being in the same segment as GC Hash, but this is not very important.

*** Pname char space is currently far too large for 32K litatoms; it might be about right for 64K, but we plan to dispose of it when pnames are hunked (taken as allocblocks), leaving just enough to get thru MAKEINIT.

**** Want to allow a little extra space for DTD in case we expand number of datatypes. This layout allows us to expand from 256 datatypes (8 bits) to 1536 datatypes (11 bits) before bumping into the LockedPageTable.

Further notes, June 1986 (post-Koto):

Pname char space now gone—all pnames are allocated from hunks.

Atom Hash Table address range used also for cml Character type (an immediate). With packages, atom hash table will go away eventually.

problem type: Performance
Subject: Want faster GETPROP
subsystem: microcode

GETPROP could be open coded faster. It is time-critical for a number of user functions. GETPROP would be faster if PUTPROP put new properties on the front of symbols PList instead of the back.

----
subsystem: microcode

TYPENAME is too slow, and is used by TYPENAMEP. Want primitive in microcode which is JNTYPENAMEP [alpha, beta, offset], like DTEST except jumps instead of traps if type doesn't match.

----
subsystem: compiler

Some of the initial constants and global variables can now be expanded inline for faster execution. Do stats on system, and look at GLOBALVAR references of functions which show up in profile.

****
Date: 12 Feb. 1982 8:53 am PST (Friday)
From: Moran.PA
Subject: CLOSEF problem

CLOSEF doesn't seem to work on the Dolphin in functions that work perfectly well on Maxc. My style is the following:

```
(LAMBDA (F)...(OUTFILE F)...(PRINTOUT NIL...)...(CLOSEF F)...)
```

When it gets to CLOSEF, it says that file val-of-F is not open. However, calling CLOSEALL in the break does close file val-of-F, which really was open and which did indeed get all the printout intended.

Tom

Date: 1 March 1982 7:20 pm PST (Monday)
From: Bobrow.PA

The compiler seems to me to be much too verbose. Cannot there be a flag so that

it only says things when either a) there is a free variable b) an undefined function called

Currently I scan my dribble file for these, which are most useful. But so little wheat and so much chaff.

****
Date: 19 March 1982 9:59 am PST (Friday)
From: VanLehn.PA
Subject: HELPFLG bug
To: Lispbug^.pa
cc: VanLehn
Reply-To: VanLehn

LISPX seems to be rebinding HELPFLG so that SETQ's at top level have no effect. I can't force errors to break by setting it to BREAK!, which makes debugging under errorsets damn near impossible.

Kurt

****

Date: 19 Mar 1982 1006-PST
From: Neil Goldman <GOLDMAN at USC-ISIF>
Subject: CONSTANT
To: MASINTER at PARC-MAXC

In the macro for CONSTANT, I propose that the evaluation of the form be done under ERRORSET protection, with an error causing the thing to be treated as a DEFERREDCONSTANT, just as is done when the value fails to pass the CONSTANTOK test.

neil
****
Subject: SEPRCASE/GETBRK bug
To: LispCore^
If a character X is defined as a macro in readtable Y, then (SYNTAXP X 'BREAK Y) is NIL, but X is not in (GETBRK Y). I don't know whether this is a bug or a feature, but in any case it means that SEPRCASE treats such a char as an alphabetic, and FINDCALLERS and friends will miss when the target atom is preceded by the macro character.

****
Date: 8 Apr 1982 1932-PST
Subject: SPECVARS and the compiler.
From: RBATES at USC-ISIB

The manual on page 18.21 states "Whenever bcompl or brecompile encounter a block declaration they rebind retfns, specvars ... to their top level value". This is NOT true as far as SPECVARS are concerned. The reason is the function LOCALVARS in COMP seeing that LOCALVARS are T sets SPECVARS to SYSSPECVARS without checking what the current value of SPECVARS is. This bug has been around since Sept 1976! These two functions with the coms show off the bug:

```
(FOO (LAMBDA NIL (PROG (X) (FUM]
(FUM (LAMBDA NIL (NILL X)]
(SETQ BUGCOMS '((FNS FOO FUM)
            (SPECVARS X)
            (BLOCK (FOOBLOCK FOO FUM (ENTRIES FOO]
```

/Ray
****
Date: 19 Apr 1982 0956-PST
Subject: DECLARE:
From: RBATES at USC-ISIB

I noticed that DECLARE: don't get compiled away (the function DECLARE: always get called), but DECLARE does get compiled away. This problem has been around awhile. Also that the example on the end of page 23.16:

```
(FOR X IN Y (DECLARE: (LOCALVARS X)) -- )
```

doesn't work.

/Ray
****

Date: 19 MAY 1982 1535-PDT
From: KAPLAN
Subject: \TAKEINTERRUPT skeleton on AINTERRUPT

I defined a skeleton for the \TAKEINTERRUPT macro on AINTERRUPT.

It has dummy calls to 2 primitives, one for checking whether \INTERRUPTABLE is T everywhere above its lowest binding (which is predictably NIL by the client of \TAKEINTERRUPT). I can simulate this with a stack search using a constant stack pointer, but this probably should be done at a lower level.

The other is a function for calling INTERRUPTED (I called it \CALLINTERRUPT). I don't think this can be the same as \CAUSEINTERRUPT. Maybe it is sufficient simply to branch to the keyboard context here.

I noticed that there are 2 global variables used mark that an interrupt is pending, \InterruptChar (used in the keyboard handler) and \INTCHAR, used in INTERRUPTED. \CAUSEINTERRUPT clears \InterruptChar and sets \INTCHAR. I don't quite understand this--is it temporary cause we haven't committed to WIND?

--Ron
****
Date: 28 MAY 1982 0003-PDT
From: KAPLAN
Subject: Bug in BCOMPL/BRECOPILE
To: MASINTER

I noticed that BRECOMPILE and BCOMPL setup LOCALVARS slightly differently.

BCOMPL initializes it to SYSLOCALVARS. BRECOMPILE does that only if LOCALVARS is T. If it is not T, it sets it to (UNION SYSLOCALVARS LOCALVARS).

Do you understand this? Which is correct, or are both correct?

If the UNION makes sense, should it also happen for SPECVARS?

--Ron
****
Date: 15 JUN 1982 2210-PDT
From: MASINTER.PA
Subject: INSPECT scrolling
To: lispsupport

If you put up an inspect window, and then change radix (e.g., from RADIX(8) to RADIX(10)), you get inconsistant output when you scroll the window.

Larry
****
We need to have an updated suite of tests to give to the technicians for hardware checkout.

Date: 18 June 1982 9:18 am PDT (Friday)
FROM: MANN
Subject:Runmicrotest.cm/replacement

Can you send us a message about Runmicrotest.cm or a suitable replacement to use in checking out the Dorado's as we discussed the other day. We do need this to do a good verification that the machines we ship run all the emulators.
****
Interlisp-D I believe has never been verified to run the DIsrael Y-FUN test of spaghetti stack operation.

Date: 23 Jun 1982 1332-EDT
From: DISRAEL at BBNG
To: masinter at PARC-MAXC

(1) YY2 is the Interlisp version of the standard fixed point

(recursion) operator in the lambda calculus. As written in lambda calculus form, it looks like this: LAMBDA F. (LAMBDA X. F (X X)) (LAMBDA X. F (X X)). You apply it to a functional - a function that returns a function as value - and then apply the result to, say a number. So if you give YY2 the factorial functional - everyone's favorite example : LAMBDA FUN. (LAMBDA N. IF N = 0, 1, ELSE TIMES N (FUN N-1)) - you get something which when applied to 3 yields 6. It does this bit of magic by unwinding an internal lambda expression 4 times (in this case). So it simulates recursion by as much iteration as you need. Note: the definition of FACTFUN is not syntactically recursive, and if one defines FACT as (YY2 'FACTFUN), then FACT is also not recursively defined.

One can actually think of YY2 (never mind why it's called that and not Y) as the limit (the least fixed point) of an infinite series, starting with Y-0 (which see) and getting on by applying Yn to G to get Yn+1. (Moreover Yn = (G 'Yn) - so every Y is a fixed point of G).

Z is another, slightly unstandard recursion operator - written in lambda calculus form as follows: LAMBDA F. ((LAMBDA X. F (LAMBDA Z. (X X) Z)) (LAMBDA X. F (LAMBDA Z. (X X) Z)). Again Z of FACTFUN is a non-recursively defined FACTORIAL applicable to numbers.

(3) As for COMBOY - it's the Y-type recursion operator written out purely in terms of the two (so-called) primitive combinators. K (LAMBDA X. (LAMBDA Y. X)) and S (LAMBDA X. (LAMBDA Y. (LAMBDA Z. ((X Z) (Y Z))))). But there is a bug in the code: as it stands, it is not applicable to numbers - only to functions; e.g. not to 3 but to LAMBDA.() 3 - and of course TIMES, SUB1, etc. barf at these.

(4) Speaking of combinators; BB is functional composition (in disguise), SKIAPPLY is function application and SKIAPPLY2 is "APPLY" - again in disguise. (It takes a function and an arg as arguments; SKIAPPLY takes a function and returns a function which is the argument function to SKIAPPLY primed for application. (Baroque, eh??). WW takes a function and produces a function which when applied to an argument, produces a version of a two-placed function whose two arguments are identified. (So SQUARE is WW applied to TIMES.)

(5) F and J are weird functionals of purely theoretical interest (unlike the others which are, as you'll surely allow, of immense practical import). J is a function provably equal to I (LAMBDA X. X); but which is, unlike I, provably non-normalizable. I, moreover, is provably the only fixed point of F. (I think J, like COMBOY, may require functions as arguments all the way down.)

To TEST:

(APPLY* (YY2 'FACTFUN) 3) will do nicely to compute (factorial 3). (The same goes for (APPLY* (Z 'FACTFUN) 3).) You can go (APPLY* (FACTFUN 'FACTORIAL) 3) - where FACTORIAL is the regular recursively defined factorial function. And, since YY2 (or Z) are fixed point operators (so F = YF) you can go (APPLY* (APPLY* 'FACTFUN (YY2 'FACTFUN)) 3). ETC...

****
Date: 27 June 1982 5:53 pm PDT (Sunday)
From: vanMelle.PA
Subject: incompatible changes

incompatible changes for whenever we feel like introducing an incompatible change:

Rearrange InterfacePage so that IFPFaultHi is even-aligned.

Rearrange DataTypeDescriptors so that DTDFREE, DTDCNT0 and DTDNEXTPAGE are all in the same quadword.

Make htfind xor the hiloc of the datum when computing the hash probe.

****
Date: 30 JUN 1982 0755-PDT
From: SPROULL
Subject: Dolphin experience

[This was a long report on Bob's experience with the Dolphin. I have exerpted the problems which I think are still relevant]

- -
The bad news

My view is that Interlisp is sinking of its own weight, and the move onto a personal computer has shown the hulk in alarming vividness. This section presents a brief justification of this view and offers possibilities for remedies. The problem can be fixed, but it may be costly.

The problem, as I see it, is that Interlisp has never had a clean internal structure of the system (as separate from the language): features and packages have accreted, wired into the existing maze to create a tighter maze. It's now so bad that a good programmer who encounters an Interlisp system is at a loss for what to do for a good long time. Few if any of the "interfaces" in Interlisp correspond to things he recognizes from other environments. He has to seek out facilities one by one; his intuition for where to look is often wrong; and he remains worried about deep interactions among various parts of the system.

This problem has been made worse by the move onto a personal computer. I think to a great extent, new facilities are added to Interlisp using the same rather low standards of interface definition that have characterized Interlisp so far. For example, while I think the facilities provided by the Interlisp-D stuff for graphics are mostly OK, the interfaces can be substantially improved.

I feel the "system" part of Interlisp needs a thorough overhaul. I favor the "open system" approach in which a very few low-level primitives are built in, and the rest is done with packages that can be separated and that have well-defined interfaces.

I realize that an undertaking such as this is a big one. However, I believe that a great deal of the detailed functionality of Interlisp can be retained (even much of the code can be retained), but the interfaces need to be redesigned in light of the tremendous evolution of the system. It's remarkable that they've remained useful as long as they have, but they need overhaul. I see two hopeful signs:

1. To the extent possible, new work should be done in the form of LISPUSERS packages. I'm delighted, for example, to see the new editor done in this way. (But I suspect the interface between it and the rest of the system could be improved if the system were improved.) I think it's important that some (perhaps all) of these packages be distributed in source-file form so that users can actually understand what they do.

2. The new Common Lisp effort is an opportunity to redesign

many of these functions. Perhaps a long-range plan might be to put Common Lisp up on the Dolphin. Should the Xerox group be contributing to the design of Common Lisp, especially in those areas where it has the most expertise (e.g., user I/O)?

Another approach is to invest some effort in restructuring Interlisp. I think it might be worth a few days' effort to estimate the difficulty of this problem and the improvements that could be reasonably expected.

(long paragraph)

To summarize, I'd like a complete, consistent "graphics package" at the low level. Some of the pieces are there (bitblt, line, etc.), but I don't see the structure. It appears to be a complex set of stuff with complex inter-relations. There is no clear description of what a bitmap is, or what a display stream is, or what a window is. This needs to be cleaned up.

More to the present point, however, is that I find the manual almost completely lacking is discussion of concepts. To pick an example from the current manual, consider the file package. What is a "file" from the point of view of Lisp? What is its purpose? What does it contain? What is the distinction between what is remembered inside the Lisp virtual memory and what is retained on the disk? And so on and on . . . While the file package might have once started out so simple that none of these questions arose, it's long past that point now.

The new graphics stuff definitely needs a good deal such concept documentation. In the manual, old and new, there's too much emphasis on functions and not enough on structure and concepts.

The best manual would result from a system restructuring The Interlisp language should form a distinct part of the manual -- some sections
of the current manual are salvageable in this respect. If the interface between the language and the packages were cleaned up, this section would describe the interface. I, for one, would benefit enormously from a self-contained section that describes the language without reference to any of the system stuff.

****
Date: 8 July 1982 8:25 am PDT (Thursday)
From: VanLehn.PA
Subject: main data space overflow
....
I've tried to find the storage leaks using COUNTDOWN, MAPATOMS and so forth. So far, the only circularities I've come across are on LISPXHISTORY. I need better leak-finding tools. One would be to do the mark & sweep part of garbage collection, including the freelist as "accessible" during the mark. The list that the sweep delivers is therefore all the cons cells that got leaked. By looking through the ones with atomic cars and cdrs, I could probably figure out from the pnames where the leaks came from.

4.   Of course, having found all the lost storage, it could be put back on the freelist, saving me a reload (but probably still taking 20 minutes). Since there is plenty of array space around, the mark & sweep could be written

simply, using its own array to hold the mark bits. I don't see any reason to do the "copy" part of a "stop & copy" since swapping doesn't seem to be a problem on the Dorado (according to temporal intuition and control T).

5. If the mark's bit array is the screen bitmap, one could probably learn alot about the storage use and maybe the chronology of the leaks by seeing not only where in vmem the leaks are, but watching the mark propagate out from specific atoms. A quick calculation has it that marking the current Mds, assuming it's mostly cons cells (95% in my program), would take a 900 by 900 bitmap.

I'd be willing to help code such a tool, or any other.

Any tools or diagnostic ideas would be welcomed with extreme enthusiasm.

Date: 8 JUL 1982 1155-PDT
From: ROACH.PA
Subject: DEFINEQ and MACROs
To: LISPSUPPORT
cc: ROACH

Dear Interlisp Support,

I think MACROs, DEFINEQ, and the Interlisp compiler interact incorrectly. I would like to define macros such as DEFEXPR, DEFFEXPR, etc. that can appear in files and will expand out into DEFINEQ forms which go on to be compiled like other DEFINEQ forms. I've been told by Ronald Kaplan that this won't work, and in fact, it doesn't. What is Interlisp's problem? I might point out that Maclisp does allow you to do this sort of thing. This is a pretty serious deficiency on Interlisp's part.

Secondly, instead of compiling expressions in the order in which they occur, the Interlisp compiler gathers functions definitions into a separate group from all other expressions. This is also a bug. (Again Maclisp does the right thing.) Compiled forms ought to load in the same order in which the uncompiled forms loaded.

I am hopeful that action will be taken on these problems.
****
Date: 27 JUL 1982 0935-PDT
From: KAPLAN.PA

. . .

We probably also ought to implement a device info interface that could, for example, tell how many pages are left on the disk, in an ifs directory, or in the ifs as a whole.

****
00348 00024 UU
Date: 10 Aug. 1982 8:48 am PDT (Tuesday)
From: VanLehn.PA
Subject: MARKASCHANGED
To: lispsupport
cc: VanLehn

MARKASCHANGED apparently doesn't inform the compiler that the function needs to be compiled. Reference manual doesn't specify whether it does or not. Obviously, it should tell the compiler to recompile.

****
Date: 26-AUG-82 11:46:38 PST
Subject: AWFUL CODE FROM CREATE USING
To: LispSupport

(create FOO using X) when FOO is a RECORD translates as a forest of CONS of CARs of CDRs rather than COPY. Produces awfully large chunks of code that doesn't even run fast. Bug?

****

Date: 8-Sep-82 14:03:40 PDT (Wednesday)
From: Masinter.PA
Subject: Re: RAISE for files

Users want to be able to read a file in lower case as if it were in upper case.

Why don't we put a translation table into READ tables? We already have them for FILEPOS and FFILEPOS etc. This would make a lot of sense. The cost is relatively small.

Date: 9-Sep-82 16:18:21 PDT (Thursday)
From: Masinter.PA
Subject: RESETLST vs RESETFORM
To: Bobrow
cc: LispSupport, Masinter.PA

This should be in the manual, or maybe we should fix RESETFORM.

Example: this doesn't work:
(RESETFORM (DEFPRINT 'A 'FOO) stuff)

This is what DOES work:

```
(RESETLST (RESETSAVE NIL (LIST 'DEFPRINT 'A (DEFPRINT 'A 'FOO)))
         stuff)
```

****

Date: 10-Sep-82 8:48:59 PDT (Friday)

There sentiment for making TY not elide comments

Date: 14 Sept. 1982 9:41 am PDT (Tuesday)
From: JonL.pa

I've never used TY, but if it does the obvious thing, then one might expect that TY* would be the command which doesn't elide comments (e.g., PF and PF*?)

****

We need to have an inbound CHAT server so you can CHAT to a machine from a remote terminal over the PUP and NS Ethernet.

We need to handle UNIX filenames better
We need device synonyms and pseudo-devices
We need to be able to delete 1100 {DSK} files without building the whole map

We need to document the facilities for doing Binary i/o for bitmaps, integers, floats. AOUT/AIN.

Compiler: [14 NOV 1981 1815-PST] (FUNCTION (LAMBDA --)) expression used as a value to be stored into a record slot. The compiler did produce a suitable subfunction, but then compiled as the value of the (FUNCTION --) expression the free varaible NEWVALUE.

We need to fix Sandbarring

Storage management:
We need to fix the GC so that it collects items which are only held as keys of hash-tables. (CLISPARRAY in particular).

we need unwindprotect

many system functions have Names which can conflicts with user fns

We need to unify the handling of meta, blank keys

we need to clean up GLOBALVARS situation

We need more diagnostics and system tests

We need to fix it so that expanding macros doesn't take so long

* FUNARG doesn't work
* MASTERSCOPE:
WHO USES FILE FREE causes funny error messages

Date: 16 SEP 1982 1802-PDT
From: BURTON.PA
Subject: compiler bug

The bind merge optimzation getts carried away with the function DSPDESTINATION on <LISPCORE>WIND>LLDISPLAY and binds the variable \INTERRUPTABLE with the variable DS. The effect is that the \DTEST is called in a context that is uninterruptable leading to a call to RAID when DSPDESTINATION is given a bad argument such as (DSPDESTINATION 123 (DSPCREATE)).

Date: 20 SEP 1982 1954-PDT
From: SHEIL.PA
Subject: Compiler bug - EVERY

Attempting to compile the expression (EVERY xxx (FUNCTION ATOM)) generates the compiler warning message (ATOM: Too many args for macro). Code seems to be OK but the message is distracting, especially if the code has come from a type? from either a record or DECLtype.

Beau

Date: 24-Sep-82 8:47:08 PDT (Friday)
From: Masinter.PA
Subject: Re: Problem with open leaf files -- and patch
In-reply-to: BOBROW's message of 21 SEP 1982 1517-PDT
To: LISPSUPPORT
cc: Bobrow, Stefik

Danny's patch to FINDOPENFILE seems to get around the immediate problem, but some more permanent fixes are needed. Ron and I talked about these problems for a while; I thought I would send out some notes on our conversation and some additional thoughts.

There are currently three separate problem areas in the current system:

a) READ/PRINT given file names which are not fully qualified scans the directory every time, which is TERRIBLY SLOW
b) for LEAF files, the file CANNOT BE FOUND by INFILEP/OUTFILEP/FINDFILE if it is already open
c) There are a number of inconsistancies having to do with the use of

DIRECTORIES and the error mechanism to implement search paths.

Proposals:
a) files which are presented to READ/PRINT (i.e., in a context where an OPEN file is required) will ONLY scan against the set of files which are open with appropriate access.

This is a change from Interlisp-10 semantics, where if you do an INFILE(FOO), and then create a NEW VERSION of FOO, and then do READ(FOO), you will get a FILE NOT OPEN error.

b) we should implement the notion of a "search path" device, e.g. {LISPUSERS} and {SOURCES}. The system will support assigning search paths to a device (new function), so that one can say {SOURCES} = {PHYLUM}<LISPCORE>WIND> , {PHYLUM}<LISPCORE>SOURCES>

Doing an INFILEP on {SOURCES}xxxx will return a full filename of {PHYLUM}<LISPCORE>WIND>xxxx, i.e., the name returned will be fully qualified. OUTFILE on {SOURCES} will write on the FIRST directory on the search path.

The general idea here is to take what is currently done via the error mechanism and DIRECTORIES and FINDFILE and instead build it in in at a lower level. This will allow some more rational implementations of the facilities.

It will also make more logical the link between the "connected" directory and the search path; that is, one can either connect to {SOURCES} or to {WIND} or to {LISPUSERS}. It will remove the distinction between FINDFILE and INFILEP in the non-spelling-correction case.

Finally, all of this is relatively easily implemented in Interlisp-10! Interlisp-10 already supports a (undocumented) feature where if you PUTPROP(LISPUSERS DIRECTORIES (<LISPUSERS> <LISP>)) and attempt to FINDFILE(LISPUSERS:filename), it will in fact search those directories.

Comments?

Date: 24-Sep-82 8:49:48 PDT (Friday)
From: Masinter.PA
Subject: Re: Bitmap editor
In-reply-to: SHEIL's message of 21 SEP 1982 1603-PDT
To: SHEIL
cc: burton, lispsupport

I always wanted to do EDITBM on a "window".

One general way of handling this is to have a general "coersion" function which coerces to "clipped bitmap", with the relatively obvious coersions for windows/displaystreams/bitmaps/regions....

Date: 24 Sep 1982 1538-PDT
From: Friedland
Subject: interlisp d bug
To: cschmidt, rindfleisch

renamefile on {DSK} doesn't work. If you have a file A 100 pages long and a file B 200 pages long and (RENAMEFILE A B), you end up with a file B 200 pages long, its first 100 being the old A and the last 200 being garbage from the old B. You have to

(DEFILE B) before the renamefile. This despite what the manual says.

PEter

Date: 29 SEP 1982 2012-PDT
From: JONL.PA
Subject: Undefined Function

Once in a while, I mistype a DEFINEQ, and wind up with an s-expression in the definition cell of some litatom which is *almost* what I wanted -- it just lacks the word LAMBDA. The error message you get when you try to run such a function is "Undefined Function" -- wouldn't it be better to reserve that msg for the case of a definition cell which is either NIL or NOBIND, and print something more informative for the case where it contains something like ((X) (LIST X (TIMES 2 X)), or (() (PRINT 5)).

Date: 29 SEP 1982 2016-PDT
From: JONL.PA
Subject: PUNTing a broken compilation
To: lispbug^

1) There needs to be an advertised way to do a BCOMPL which ignores errors which occur during the compilation of a single funciton, so that a single call to BCOMPL will proceed thru the while file, finding perhaps other errors before ending.
2) I tried the unadvertised function (PUNT) after one of my macros caused a BREAK during compilation, and it appeared as though the then-current break window became the normal TTY display stream.

Date: 30 SEP 1982 1119-PDT
From: SHEIL.PA
Subject: two comments on the RESETFORM macro

Currently, the RESETFORM macro evaluates the resetform (a) outside of errorset protection and (b) some time (during which an interrupt can occur) before the resetlst entry for undoing it is made. This could be disasterous if one is resetting, for example, a display stream clipping region and the user bombed you out. [(a) may or may not be a bug or non-feature, depending on how one reads the manual; (b) is nasty to fix and probably requires interrupt protection]

Also, the RESETFORM macro doesnt do a very good job if one passes it a LAMBDA expression as the reset function (two copies wind up in the compiled code). [The motivation for this is for two arg fns like DSPCLIPPINREGION, as

```
      (RESETFORM ((LAMBDA (X) (DSPCLIPPINGREGION X window)) NEWREG)
            forms)
is much more elegant than
      (RESETLST (RESETSAVE NIL (LIST 'DSPCLIPPINGREGION
                              (DSCLIPPINGREGION NEWREG window)
                              window))
            forms)
```

In fact, since I just realized that, it might be worth noting this trick in the
manual for other slow thinkers.]

*** I really want to shift to this notation in DEDIT, so this patch would be greatly appreciated ***

Beau

PS: From a slightly broader point of view, it would be nice to have a wizard scrutinize these macros as (a) this is not the first non-feature report for them (b) they dont look to be as good code as they could be. Last time I raised this, the discussion quickly expanded to include respecifying the whole error handling machinery (and thus nothing happened). Perhaps a useful intermediate step would be to define a few more useful abstractions, such as CATCH and THROW, which we could start using in our code to replace the convoluted RESETLST constructions that tend to generate these discussions in the first place.

****

Date: 1 OCT 1982 1430-PDT
From: SHEIL.PA
Subject: Glitch in RENAME
To: LISPSUPPORT

If one has a variable FOO which is used in some file BAR and you wish to rename
it to FUM, (RENAME 'FOO 'FUM 'VARS 'BAR) will bomb with complaint "no VARS defn
for FOO" unless FOO has a top level binding.

Beau
****
00705 00024 UU
Date: 1 OCT 1982 1611-PDT
From: SHEIL.PA
Subject: PP and PRETTYPRINT glitches
To: LISPSUPPORT

Some time ago, PP (and PP* and PPT) had LOCALVARS declarations added so that their variables would not interfere with EVALVs from PRETTYPRINT. Unfortunately, the ERRORSET implementation causes all these to be SPECIAL in Interlisp-D anyway. Pending a more general resolution of this problem, it would be nice if these fns were patched to avoid the fact that (PP X) for example, does absolutely nothing. [Major motivation: This is irritating if you know what is going on but absolutely inexplicable if you dont].

Manual note: ARGLIST of PRETTYPRINT does not match manual spec.
****
Date: 3-Oct-82 21:05:18 PDT (Sunday)
From: Masinter.PA
Subject: Re: PP and PRETTYPRINT glitches
In-reply-to: SHEIL's message of 1 OCT 1982 1611-PDT
To: SHEIL
cc: LISPSUPPORT

An additional (more general) fix is for the compiler to rename the variables which are auto-SPECVARed because of the ERRORSET hack.
****
Date: 3 OCT 1982 2327-PDT
From: KAPLAN.PA
Subject: Clispify/dwimify bug

If (don't ask why) the atoms < and > have top-level values, then CLISPIFY((LIST (FOO))) is (< (FOO) >) which then doesn't dwimify back.

As a minimum, this (and all) clisp transformations should not be performed in environments where they won't dwimify properly.

****
Date: 4 OCT 1982 0514-PDT
From: JONL.PA
Subject: MASTERSCOPE Message

If you edit a macro, MasterScope will correctly tell you that

certain functions depend upon it; but when calling UNSAVEFNS,
it always prints the message "Loading FooFunction", regardles
wof whether it is really loading it, or merely UNSAVEDEFing it.
****
Date: 4 Oct. 1982 8:26 am PDT (Monday)
From: Stefik.PA
Subject: Re: Interlisp-D planning

Larry and Bill, -

The Lisp features {desired by Mark et al.) were

(1) Ability to have functions without naming them.
(2) Ability to hash on strings (or unintered atoms?) for our LOOPS uids.
(3) Access to an efficient (say B-tree) database (perhaps cedar?).

Of more immediate importance will be some participation by yous guys in design reviews of LOOPS, and consulting on performance tuning. Mark

****
Subject: lisp.tasks
* UNBREAK work on internal block functions just like TRACE
* DUMMYFRAMEP definition correction
* compiler optimization NEWOPTFLG=T
* free code deleterefs pointers therefrom

Interlisp-10 problems
* DELFILE BUG
* GETSTREAM
* CALLSCCODE returns duplicate values
* add ALLOCSTRING
* make NCREATE allocate system datatypes too

Date: 6 Oct. 1982 9:44 am PDT (Wednesday)
From: Bobrow.PA
Subject: Re: Interlisp-D planning
In-reply-to: Masinter’s message of 5-Oct-82 19:45:57 PDT (Tuesday)
To: Masinter
cc: Stefik, Bobrow, vanMelle

1) Must function call always go through an atom? Perhaps the name slot on the stack could be made in the "naked" case to point to the fn data object. Inspect
macros might allow at least finding out about arguments expected.
One might even have a string in such an object to name it (this would work for our methods).

2) The current atom hash table code made available as string hashing would do quite nicely at this stage for us, I think. More than 2^16 atoms would be nice,
but is not right, since we want separate name spaces with overlapping names, and I don’t think we need anything but string hashing.

3) Eventually we might want to use file based indexes for knowledge bases (e.g.
BTrees) when they get too large. The current Alpine project is NOT planning to
provide a BTree index interface at the moment (I checked with Mark Brown)
Most of the indexing stuff is done now on the clients side of the Cedar database.

Stats runs show that our GetValue and PutValue are remarkably slow (about 250 microseconds on a Dorado) and take most of the time, as we expected. Help on redesign on that would be most welcome.

danny

Date: 4 OCT 1982 0514-PDT
From: JONL.PA
Subject: MASTERSCOPE Message
To: LISPBUG^

If you edit a macro, MasterScope will correctly tell you that certain functions depend upon it; but when calling UNSAVEFNS, it always prints the message "Loading FooFunction", regardles of whether it is really loading it, or merely UNSAVEDEFing it.

Date: 4-Oct-82 12:53:41 PDT (Monday)
From: Masinter.PA
Subject: lisp.tasks
To: masinter
cc: , Masinter.PA

* Compiler: [14 NOV 1981 1815-PST] (FUNCTION (LAMBDA --)) expression used as a value to be stored into a record slot. The compiler did produce a suitable subfunction, but then compiled as the value of the (FUNCTION --) expression the free varaible NEWVALUE.
* Name conflicts with system fns
* EVAL edit macro in editor needs ERSETQ instead of NLSETQ.
* correct handling of meta, blank keys
* periodicallyreclaim be sensitive to mouse events, process suspension.
* Breakcheck problem associated with heavy swapping on first uba or udf.
* improve interface to stats for other things than fn call
* make STACK FULL non-fatal error
* (APPEND circular) bug
* Can create ARRAYs > 2^16.
* STORAGE) function prints garbage negative numbers in the 3rd column.
* UNBREAK work on internal block functions just like TRACE
* BRKDWNRESULTS print out
* interaction of code which rewrites filemaps and RADIX
* rename low level functions
* memory map diagnostics
* DUMMYFRAMEP definition correction
* Interlisp-10 problems:
   * DELFILE BUG
   * samedir has problems on tops20: directoryname neq filenamefield
   * CALLSCCODE returns duplicate values
   * add ALLOCSTRING
   * make NCREATE allocate system datatypes too
* Code for generating Interpress from Tops-20 and Vax (Troff, Scribe, ...)
* small demo
* LISPXSTATS
* compiler optimization NEWOPTFLG=T
* free code deleterefs pointers therefrom

Misc bugs
      Masterscope recursion with long names
BQUOTE

Date: 6-Oct-82 13:21:14 PDT (Wednesday)
From: Masinter.PA
Subject: questions & comments from Schoen
To: LispSupport
cc: Masinter.PA

document VRAID package; I think as a LispUsers package.

we should provide FLOUT and READ/WRITEBINARYBITMAP for fast i/o of floatps and bitmaps.

Schoen has a VAXPRINT package (possibly similar to VanBuers) which causes the vax to hardcopy screen bitmaps. He doesn't know what the Versatec they use is exactly, but it has 2112 dots across, he says.

Date: 8 OCT 1982 1620-PDT
From: SHEIL.PA
Subject: Lisp task list
To: MASINTER
cc: lispcore^

I just spent 15 mins reading it. Very comprehensive; fine job; don't envy you the task of prioritizing it! A couple of additonal points:

ERRORSET compilation.
    Possibility of marking the stack rather than introducing new function call. If not, the making compiler suppress or rename forced new specvars.
Error handling
    Proposal to make ^D work by ^E (incompatible but worth it?)
    Some improvement over current mess.
    Failing that, debugging optimization of RESETLST/SAVE/FORM
Global vars
    If RESETVARSLST is known not to be declared global, lets fix it rather than documenting it! Mike Sannella needs some help to get new manual to indicate which system params are global - maybe he could propose a list derived from the manual and we could dispose of this one.
Garbage collection
    Compiled code blocks (pointers therefrom)
    Hash arrays [urgent; current incompatibility + perf problem]
  File package
    Hasdef problems
    Fixing EDIT interfaces to use same.

Beau

Date: 11-Oct-82 16:41:58 PDT (Monday)
From: vanMelle.PA
Subject: CASEARRAY for READ
To: LispSupport

Yet another (perhaps the same) request for (RAISE T) for files...

Mail-from: Arpanet host SU-SCORE rcvd at 11-OCT-82 1414-PDT
Date: 11 Oct 1982 1402-PDT
From: David E. Smith <CSD.SMITH at SU-SCORE>
Subject: lower case
To: vanmelle at PARC-MAXC
Stanford Phone: (415)497-1809

I need a way of forcing interlisp to be case independent for file input as well as terminal input. Read macros won't do it because I don't want the lower case letters to be break characters and "ALWAYS" forces this. Advising or rewriting READC presumably wouldn't work either because strings and characters prefaced by "%" would then get upcased.

How do I do this? Am I forced to rewrite LOAD and all of its accomplices? Crufty and/or release dependent solutions will not be sneezed at. Help!

-- de2

Date: 12-Oct-82 13:11:06 PDT (Tuesday)
From: vanMelle.PA
Subject: Font assumptions
To: Burton
cc: LispSupport

If you do (FONTSET 'STANDARD) to turn off fonts (e.g., to make a fontfree file), subsequent calls to the inspector die in a \DTEST of FONTDESCRIPTOR because DEFAULTFONT is NIL.

I wonder how many other places make such assumptions.

Incidentally, herewith a reminder that \DTESTFAIL desperately needs to produce a better error message, at least incorporating its second arg (the intended type).

Bill

Date: 14-Oct-82 16:36:57 PDT (Thursday)
Subject: Re: Schlumberger URGENT
In-reply-to: Raim.EOS's message of 14 Oct. 1982 10:24 am PDT (Thursday)

Eric (and users in general) should avoid doing (APPLY 'IMAX LST) and instead write (for X in LST maximum X).

The limit of number of arguments to a function is indeed 80; it is possible that we could bump it, but there still would be a fixed limit.

Larry

Date: 15-Oct-82 15:05:36 PDT (Friday)
From: vanMelle.PA
Subject: LARGEST/SMALLEST
To: LispCore^

It has been pointed out that these names are confusing, due to the ambiguity of what you might want the iterative to return. I propose that LARGEST be called MAXIMIZING, SMALLEST be MINIMIZING, and that there also me oprs MAXIMUM and MINIMUM. Since FIND is a synonym of FOR, we could thus have:

```
  (find X in L maximizing (FOO X))
            returns the X for which FOO is largest, and
  (for X in L maximum (FOO X))
            returns the largest value of FOO over L.
```

Bill

Date: 23 Jan 89 16:12
From: Will Snow:AISNorth:Xerox
Subject: Places that use hiloc/loloc and what for
To: sybalsky:AISNorth:Xerox, shih:AISNorth:Xerox
cc: Will Snow:AISNorth:Xerox

I spent some time finding out who uses hiloc/loloc and what they use them for. the following is a summary:

10MBDECLS:

D0ETHERIOCB
DLETHERIOCB

All iocb's are in the lowest addresses, so 10MBdecls takes advantage and only puts in the piece of the pointer necessary (loloc)

10MBDRIVER:

\RELEASE.IOCB change the "next iocb" field of the given iocb to a null value.
\INIT.ETHER.BUFFER.POOL change the "next iocb" fields to null.
\QUEUE.INPUT.IOCB next iocb ptr of last iocb = this iocb, or make this the first one.
\QUEUE.OUTPUT.IOCB rearranging the queue of iocb's

ABASIC:

EQUAL uses loloc as an optimization on fixp vs smallp comparisons.

ACODE:

CHANGECCODE uses hiloc, loloc during refcount operations
CODEBLOCKP uses hiloc to determine what segment of storage the piece being looked at is in.

ADDARITH:

MACRO **.XUNBOX.**

APRINT:

\PRINTADDR tries to print a lisp address nicely. Uses both HILOC and LOLOC.

ASTACK:

SETSTKNAME ? (HILOC)

CMLARRAY-SUPPORT:

MACRO **%SMALLFIXP-SMALLPOSP** converts smallfixp to a number.(LOLOC)

CMLCHARACTER:

ACCESSFNS **CHARACTER** how to create and access a common lisp character. (loloc)

CL:CHAR-CODE change a character into a #(Loloc)

DEFOPTIMIZER **CL:CHAR-CODE** fast changing of char into code. (loloc)

CL:CODE-CHAR fast checking for smallposp (hiloc)

DEFOPTIMIZER **CL:CODE-CHAR** fast checking for smallposp (hiloc)

CMLEVAL:

DEFSTRUCT **CLOSURE** :print-function to print the ptr (hiloc,loloc)

DEFSTRUCT **ENVIRONMENT** :print-function to print the ptr (hiloc,loloc)

set-symbol checks if environment is the stackhi

CMLSTRING

%%STRING-BASE-COMPARE-EQUAL get the character code from a string of CL:CHARS (loloc)

CMLUNDO

undoably-set-symbol determine if at top of stack. (hiloc)

D-ASSEM

FIXUP-PTR, **FIXUP-PTR-NO-REF** - ? (loloc,hiloc)

INTERN-DCODE ? (loloc)

DEBUGGER

PRINT-ENTRY-MESSAGE print the condition number...

DLAP - LOLOC/HILOC optimizers to u-code.

DOVEDECLS:

DEFMACRO **\DoveIO.IORegionOffset** get the right IO region on a dove.(loloc)

DOVEETHER

\DoveEther.EnQueue fill in the "next packet" field.(loloc)

DOVEINPUTOUTPUT

\DoveIO.MakeOpieAddress make the correct opie address out of a lisp addr.(hi,lo)

DTDECLARE:

COMPILEDREPLACEFIELD figure out what to do with an X pointer.

Note:: Both DLION and DOVE disk code also uses LOLOC and HILOC...

—End of message—

EVERYTHING YOU WANTED TO KNOW ABOUT THE AR DATA BASE - BUT WERE AFRAID TO QUERY

This document on {Pogo:}<Release Management>Kat>Doc>AR-How-To-Edit.tedit.
Last edited by Kat Kohlsaat on 9-Mar-1988 15:48:55

INTRODUCTION

The Action Request data base is the primary vehicle through which the state of Xerox Lisp, including outstanding problems, requested features, and the like, is tracked. Since ARs are the primary channel of communication between the user, customer support, marketing, and development, it is important that the maximum amount of correct information be compressed into each AR. This allows technical information to get to development, and just as importantly, get back out. This process can be facilitated by correct use of the fields of the AR.

THE AR FORM - THE FIELDS AND WHAT THEY MEAN

The basic component of the AR data base is the individual AR. An AR is the melding of a blank AR form with the data specifying a need. The AR form provides 31 areas, or fields, for the input of information giving a concise summary of the need. A need can be either a problem with the Xerox Lisp system that must be corrected, or a request for a feature that would improve the system if implemented. Since the structure of the AR form must be standardized to allow entry of a wide variety of needs, the data detailing the needs becomes an important component of the AR system. Correct use of the various fields comprising an AR facilitates the exchange of information between the submitter of the AR and the developer who will act upon the AR.

The FillInDefaults option of the left button menu associated with the AR Bug Report Editor title bar will fill in the **Submitter:**, **Source:**, **Status:**, **Machine:**, **Microcode Version:** and **Memory Size:** fields, and will place `MAKESYSNAME` as well as `MAKESYSDATE` in the **Lisp Version:** field. Please fill in the **Lisp Version:** field when submitting an AR, either by typing it in or by using FillInDefaults from the pop-up menu. The version of software the bug is being found in is important data.

Number: Generated by the AR data base, every AR has a unique number. AR numbers are *never* recycled. ARs are never deleted. The AR number cannot be changed by the user.

Date: The date the AR was originally submitted. This is filled in by the system.

Submitter: The login name of the person who submitted this AR. This is filled in by the system.

Source: The name of the person reporting the problem being documented in the AR. The name or names appearing in this field must give enough information to enable contact if needed, i. e., `Doe.PASA` or `Doe@Berkeley.edu`.

Subject: A terse summary of the problem, providing both enough information to identify it uniquely and enough keywords for querying. "`FOO doesn't work`" or "`Floppy problem`" is not good enough. Think of yourself as a newspaper headline writer: "`Attempt to write file when floppy door is open causes awful noise.`" Implementors may change the **Subject:** field as more details about the true nature of the problem become apparent.

As much as possible, relevent keywords should be included in the **Subject:** to facilitate querying on the data base. If the problem relates to a specific package, that package name should be mentioned in the **Subject:**. File names, commands, functions, error messages, etc., are good examples of relevant keywords. For example, rather than "`Floppy breaks when using mailfile`", a better subject would be "`Loading mail file from floppy causes break in FLOPPY.OPEN with error ILLEGAL ARG: 42`."

Assigned To: The name of the person or persons who took some action based on the AR.

Attn: The name of the person or persons responsible for fixing the AR.

Status: This field shows the status of the AR. This changes as action is taken on the AR.

New — All ARs are generated with a default Status of *New* when submitted. *New* ARs have not been reviewed.

Open — This reviewed AR describes an outstanding problem with released software.

Open/Unreleased — This reviewed AR describes a problem with unreleased software.

Fixed — Problem has been fixed in an Internal loadup. Developers marking ARs as *Fixed* should mark the **In/By:** field according to the release into which the fix is being incorporated. At this time, the developer should also fill in the **Release Note:** field.

Closed — System with fix in it has been tested, documented, & released.

Declined — ARs can be declined for any of a variety of reasons. Perhaps it’s a request for feature that is officially "never" going to be implemented (e.g., we think it’s a bad idea). Perhaps the bug report is considered spurious (development doesn’t think it is a bug). The reason for the AR being declined should be included in the **Description:** field. *Declined* ARs will be reviewed periodically so that old ARs may be re-opened.

Superseded — Another AR already includes the problem described in this one. The **In/By:** field of the superseded AR should include the AR number of the one that supersedes it (ex., 7064), and the begining of the **Subject:** field should be edited to include a notation such as: "Superseded by AR #7064". The superseding AR should contain the information contained in the AR it supersedes, with a notation in the **Description:** such as: "[Supersedes AR #7911.]".

Obsolete — The problem reported is no longer a problem, e.g. the module containing the reported problem is no longer supported.

Incomplete — The information submitted is not enough to take action, i.e., there is not enough information to identify the bug, or the feature request doesn’t give enough detail about what is wanted. This is different from *Declined* in that the request is considered valid, but the AR remains open awaiting more detail.

Internal — This status is used to report problems with internal software.

Wish — This status is usually used to request new features, change of features, Design-Impl or Design-UI .

Problem Type: Defines the type of problem described in the AR. Possibilities for the **Problem Type:** field follow:

Bug — The system does not work as documented.

Design-Impl — The system works, but the internal implementation is wrong. (This type is generally submitted by other developers.)

Feature — Used to indicate a feature request.

Design-UI — The design of the user interface is wrong. This includes problems in the way in which things display, as well as program callable structures.

Documentation — The system works, but the documentation is wrong, unclear, or incomplete. The **System:** and **Subsystem:** fields should reflect the area in which there is a problem with the documentation. The **System:** should <u>not</u> be *Documentation* unless there is a specific problem with the documentation, apart from the system, e.g. "`need better index`".

Performance — The system works, but it is too slow doing the described operation.

Difficulty: A rough estimate of the difficulty of the problem. *This field is to be filled in by developer only.* Categories within **Difficulty:** follow:

Easy — < 1 week to fix

Moderate — < 1 month to fix

Hard — < 6 months to fix

Very Hard — > 6 months to fix

Impossible — can't be fixed

In/By: Used to specify the release for which an AR is/will be fixed or to indicate the number of a superseding AR.

Impact: How seriously does it affect your ability to get work done, value of Xerox Lisp, etc. The items apply to bug reports, but feature requests should be rated along analogous lines. The categories within **Impact:** follow:

Fatal — Causes the system to crash, causes a loss of work, etc. Problem resolution is a requirement for project completion.

Serious — The problem can be worked around but it seriously interferes with work. This type of problem usually requires substantial reimplementation.

Moderate — The problem is tolerable, but clearly a problem, and the responsibility of Interlisp development.

Annoying — The problem is annoying, a minor request for a new feature that "would be nice".

Minor — May be some dispute about whether it is even a bug, or a very minor feature request.

Frequency: How reproducible is the problem? If it is not known or is irrelevant to the AR, leave it blank. This is generally only relevant for bug reports. **Frequency:** can be one of:

Everytime — Reproducible every time.

Intermittent — Doesn't always happen.

Once — Saw it happen once.

Priority: The perceived priority of this problem relative to the next release. A submitter may fill in their desired priority when submitting the AR. *However*, *priorities are approved/changed **only** by the Change Control Board.* Four different priorities are possible:

Absolutely — A showstopper. The pending release will be held if this AR is not completed. Requirements for this rating are: 1) Work lost with no workaround; 2) Highly embarassing to Xerox; or 3) Marked *Hopefully* for previous release.

Hopefully — Preferable to be in the pending release, otherwise will be in next release.

Perhaps — Will get implemented if other revisions in same area are completed.

Unlikely — Unlikely to be included in the next release.

System: Subsystem: The category and sub-category of the Xerox Lisp system that is pertinent to this AR. **System:** and **Subsystem:** categories are:

Communications

- NS Protocols
- NS Filing
- NS Printing
- PUP Protocols
- PUP FTP
- Grapevine
- Leaf
- RS232
- VAX Server
- DEI
- EVMS/RPC
- Lisp Servers
- Clearinghouse
- TCP/IP
- Centronics
- TTYPort
- Chat

	Chat Interface Pup Chat Driver NS Chat Driver RS232 Chat Driver TTYPort Chat Driver Chat DM2500 Emulator Chat VT100 Emulator NSMaintain Other
Windows and Graphics	Window System Library Fonts Printing Color Bitmaps Demos Menus Other
Operating System	Virtual Memory Generic File Operations DLion Disk Daybreak Disk DLion Floppy Daybreak Floppy Dolphin/Dorado Disk Processes Streams Keyboard Mouse Other
Language Support	Arithmetic Compiler, Code Format For/If Microcode Storage Formats/Mgt Garbage Collection Read and Print Stack and Interpreter Bootstrapping and Teleraid Diagnostics Other
Programming Environment	Break Package Code Editor DWIM Inspector File Package History Masterscope PSW Record Package Performance Tools Edit Interface Exec Presentations

	Stepper Other
Text	TEdit TTYIN Lafite AR Database Other
Common Lisp	Type System Declarations Macros Control Structure Evaluator Symbols/Packages Arithmetic Characters/Strings Sequences Lists Arrays Structures Hash Tables Streams and I/O File System Interface Error System Compiler Tamarin Support Microcoded Operations Common Loops Other
CLOS	Language Browsers Methods Classes Meta Classes Other
Port	Other
Maiko	Bytecode Emulation Native Code I/O System Host Integration Host User Interface Foreign Fn Interface Installation Procedure Documentation Other
LOOPS	Active Values Composite Objects Objects Browsers User Interface Virtual Copy Other

PCE	Monochrome Display Color Display Keyboard Emulated Rigid Disk Floppy Disk Printer Port User Interface Programmatic Interface File System Interface Memory Ethernet Configuration Tools Other
PROLOG	Arithmetic Dinfo Microcode Editor Interface Compiler Interpreter I/O Debugging Prolog-Lisp Interface Other
4045	XLPStream Remoteserver HQStream PSO Other
Rooms	Window Types Overview Suites Buttons Documentation Other
Library	Cash-File CharCode Tables Copyfiles DEdit DatabaseFns FX-80 Printer Support Filebrowser Font Samples GCHax GraphZoom Grapher Hash Hash-File Image Object Interface Kermit Masterscope Browser MatMult Press Printer Support SameDir Sketch

	SysEdit/EXPORTS.ALL Tablebrowser Virtual Keyboards Where-Is Other
BusMaster	Speech Color Other
Documentation	Tools 1108 Users Guide 1186 Users Guide Primer Product Descr/Tech Summary Hardware Installation Guide Programmers Introduction Interlisp Reference Manual Library Package Manual Internal System Documentation Other
Other Software	Installation Utility Release Procedure Other

Machine: Disk: The value of these fields should be the type of Xerox hardware that is pertinent to this AR, i.e., the machine and disk on which the problem is happening. **Machine:** and **Disk:** categories are:

1108	SA1000 (10 MB) SA4000 (29 MB) Q2040 (43 MB) Q2080 (80 MB) T80 (80 MB) T300 (300 MB) Other
1132	T80 (80 MB) Century315 Other
1186	ST212 (10 MB) TM703 (20 MB) TM702 (20 MB) ST4026 (20 MB) Q530 (20 MB) Q540 (40 MB) Micropolis 1303 (40 MB) Micropolis 1325 (80 MB)

Lisp Version: This field should identify the Xerox Lisp sysout in which the problem occurs (or the feature doesn't occur). The sysout should be identified by the name associated with the release (Koto, Lyric, Medley, etc.,) and/or `MAKESYSDATE`.

Microcode Version: This information may be found by typing `(MICROCODEVERSION)` in an Interlisp Exec or `(il:microcodeversion)` in a Common Lisp Exec.

File Server: What type of file server, if any, is involved with this problem. The menu contains the following items:

8037
IFS
NS
VAX/VMS - 3 MB
VAX/VMS - 10 MB
VAX/UNIX
Micro VAX/VMS
Other

Source Files: The source files pertinent to the problem being reported in this AR.

Memory Size: This value is the amount of "real memory", or RAM, in pages. This information may be found by typing `(REALMEMORYSIZE)` in an Interlisp Exec or `(il:realmemorysize)` in a Common Lisp Exec.

Server Software Version: The version of software running on the server.

Disposition: The record of who has changed which fields of this AR and when it was done. <u>This is filled in by the system.</u>

Release Note: This field should contain the information to be included in the Release Notes for a given release. It should be release specific, such as: "Medley: In the debugger, the frame inspector window . . . " If a release note isn't required, that should also be explicitly mentioned, example: "Lyric LOOPS: None needed."

Description: This field should contain a complete description of the problem or request, including any subsequent discussion after the AR submission. If the bug report came via electronic mail, the entire report should be added into this field. In cases where there are a number of electronic mail messages discussing this problem, all messages should be appended into this field.

Workaround: This field should contain a known procedure to work around the problem until it is fixed. This would generally be a short recipe.

Test Case: This field should contain a list of the files needed to recreate the problem. Please note that any Common Lisp or Interlisp recipes for reproducing the problem should be in the **Description:** field, not in the **Test Case:** field. When the problem is *Fixed* the **Test Case:** field should include any appropriate information that can be used to confirm the fix (or a note that a Test Case is not applicable, ex. "N/A").

Edit-By: The login name of the last person to edit the AR. <u>This is filled in by the system.</u>

Edit-Date: The date of the last change made to the AR. <u>This is filled in by the system.</u>

WHAT HAPPENS TO AN AR AFTER IT IS SUBMITTED?

Change Control Boards have been established for each XAIS product to bring AR priorities more in line with customer needs. The membership of each Change Control Board consists of the Product Development Project Leader, a member of Customer Support, and a member of Release Management. Incoming ARs are reviewed weekly by the appropriate board. At this meeting priorities are assigned for each AR, and other pertinent information, such as who will deal with the AR, is gathered. This information is input to the AR data base and summaries of ARs are generated for each responsible developer.

When a problem is resolved, the **Status:** field of the associated AR is changed to *Fixed*. At this point, the software is incorporated into the Development environment, which is the precursor for the next release. The *Fixed* AR is sent to the Documentation group for incorporation into the appropriate part(s) of the product documentation. When a release of software to customers occurs, all *Fixed* ARs that have been incorporated into that release, software and documentation, are marked *Closed*.

Array Space

Misc Notes

Date: 3 Apr 89 11:16
From: vanMelle:PA:Xerox
Subject: Re: Number of Locked pages?
In-Reply-to: Ingalls.wbst's message of 2 Apr 89 19:40 EDT
Reply-to: vanMelle:PA:Xerox
To: Ingalls:WBST:Xerox
cc: LispFolklore^:X:RX

(\COUNTREALPAGES 'LOCKED) is what you want. Other interesting arguments to \COUNTREALPAGES include DIRTY, REF (pages that have been referenced since the last sweep of the real page table) and OCCUPIED (pages inhabited by Lisp vmem pages, as opposed to things like the Alto emulator (Dorado), the real page table and low-level system buffers, thus less than (REALMEMORYSIZE)).

Bill

((0 246) (1 15332) (2 3591) (3 1765) (4 897) (5 654) (6 442) (7 339) (8 232)
(9 180) (10 176) (11 125)
(12 108) (13 107) (14 91) (15 65) (16 74) (17 48) (18 57) (19 46) (20 35) (21
31) (22 28) (23 28) (24
24) (25 22) (26 26) (27 27) (28 9) (29 18) (30 10) (31 9) (32 10) (33 7) (34
10) (35 6) (36 4) (37 8)
(38 2) (39 4) (40 6) (41 7) (42 3) (43 11) (44 4) (45 3) (46 3) (47 4) (48 5)
(49 1) (50 3) (51 2) (
52 2) (53 1) (54 1) (55 4) (56 7) (57 2) (58 2) (59 4) (60 2) (61 1) (62 3)
(63 2) (64 2) (66 1) (67 5
) (69 1) (70 2) (72 1) (73 1) (74 1) (75 2) (77 2) (79 1) (82 2) (83 1) (84 2)
(86 1) (89 1) (91 1) (
93 2) (97 1) (111 1) (116 1) (119 1) (130 1) (133 1) (136 1) (145 1) (146 1)
(148 1) (152 2) (159 1) (
163 1) (168 1) (179 1) (185 1) (188 1) (196 1) (207 1))

Subject: A proposed design for big bitmaps (for Maiko Color (aka Kaleidoscope))
From: shih:mv:envos

Here’s a proposed design for big bitmaps (for Maiko Color). This message is also filed as BigBitmaps.TEdit, under {Pogo:MV:envos}<DSUNLISP>Documents>Development>Color>, and {Eris}<lispcore>internal>doc>.

There are basically two alternatives, either change allocblock to allow blocks bigger than 65K, or to change bitblt (and perhaps many other functions) to allow a new datatype, BIGBM.

The following describes the second BIGBM alternative.

DESIGN MOTIVATION

The color code currently uses ALLOCPAGEBLOCK to create a non-GC’able large bitmap for the color screen. Windows on that screen, I believe point to that screen bitmap, but Window "backing bitmaps" (the thing that holds what is behind an open window, and what is inside a closed window), is a separate bitmap.

The problem is that the backing bitmap currently cannot be as big as the window would like it to be.

The BIGBM design would keep the noncollectible color screen bitmap, but would allow windows to have BIGBM backing bitmaps. In addition, this design is more likely to be portable backwards (e.g. into Medley1.0 or Medley1.1) since no existing system datatypes need to be changed (unlike the plan to change ALLOCBLOCK).

Since XAIE does not (I believe) allow (currently) DIG operations on closed windows, then there are no DIG operations (except BITBLT) which need to occur on BIGBM backing bitmaps.

Therefore, a BIGBM need merely be a GC’able datatype which supports BITBLT between itself and all other legal datatypes (principally windows and bitmaps, but possible streams (e.g. Interpress), and possible other datatypes).

Longer term, having a generalized BITBLT will be useful, for the following reasons:

Color -   8 bpp bitmaps will begin to push Medley’s 32Mb address limitation, and 24 bpp bitmaps certainly will (the screen alone will take up 3Mb = 10% of the address space!).

Remote Bitmaps - there may be applications which need bitmaps outside of the address space (color above for example). Nick Brigg’s Maple color processor for the 1186 had a generalized remote bitblt, which "did the right thing" when source and destination bitmaps were both remte (e.g. remote bitblt).

XWindow Medley- might require remote bitmaps.

NonSun Medley - might require remote bitmaps, especially because it is doubtful the SunOS system call MMAP exists elsewhere. When not, the screen itself will need to be remote.

BIGBM DESIGN

A straightforward design would be to have a BIGBM simply be a collection of ordinary bitmaps ("slices of the big bitmap"), and then generalize BITBLT to handle the collection (by repeated bitblts).

More elaborate designs would allow BIGBMs to be a collection of any BitBlt'able objects (streams, windows, bitmaps, bigbitmaps), or perhaps collections of raw blocks (to save some of the BITBLT initialization overhead).

Probably at least one dimension of the slices should be a multiple of the LCD (least common denominator) of any TEXTURE (currently 16), so that textures do not show seams across the slices.

Probably also the slices should be "short and fat" (e.g. the BIGBM is made up of several rows of bitmaps), since the ucode inner loop runs in the X direction, but this partially depends on the application.

For example, font bitmaps are short and *very* wide. If large color fonts need a BIGBM font bitmap, then each character would cut across several slices, slowing down each character. For fonts, column slices might be better. Note that each character is also more "coherent" in memory this way, which might improve paging behaviour.

Having BIGBMs be composite objects (rather than a large coherent allocblock) might also improve GC behaviour, because a BIGBM can be allocated out fragmented free space. There may not be enough coherent free space to allocate a large allocblock, due to fragmentation.

Another design elaboration would be to provide full Imagestream capabilities onto BIGBMs, to that (DSPCREATE dest) will work on BIGBMs.

BIGBM IMPLEMENTATION

There is currently a draft implementation of BIGBMs on {Eris}<lispcore>internal>library>BIGBM. It has several limitations:

0. It is intended to be a subfunction of BITBLT (which explains the limits 1 & 2 below)
1. It doesn't handle clipping (I'm not sure, but clipping might be handled generically in BITBLT by changing the srcex, srcey, destx, desty, width, & height args).
2. It doesn't handle default arguments (BitBlt does this).
3. It currently only makes the slices 16 bits tall (it should make the slices as big as possible).
4. Its not very efficient. It only walks the slices linearly (the first relevent slice could be found by algebra), and it doesn't stop when the last relevent slice is bltted.
5. Because of 3 & 4, for large bitmaps its roughly 2x slower than normal bitmap blts. For small bitmaps, it may be much worse. This may be OK though for backing color windows, because color windows are large.

Mostly, this code needs to be folded into BITBLT & BLTSHADE, at the appropriate spots, and optimized if necessary.

Abstract: This document is a general introduction to the Interlisp-D computing environment, slanted towards the needs and interests of newcomers to the Intelligent Systems Laboratory and Xerox Artificial Intelligence Systems. This is definitely in *DRAFT* form, since it still talks a lot about Cedar and hardly any about Interlisp-D!!!!

For Internal Use Only

Raison d'Etre

The purpose of this document is to help immigrants adapt to the local computing community. "The local community" primarily means Interlisp-D users within PARC's Intelligent Systems Laboratory and the development group of Xerox Artificial Intelligence Systems. Immigrants to other computing communities within Xerox may also find this document of interest no guarantees are made. I shall assume herein that said immigrants know quite a bit about computers in general. Hence, I shall concentrate upon discussing the idiosyncratic characteristics of the local hardware environment, software environment, social environment, linguistic environment, and the like. This document was "ripped off" from a similar one written for the Computer Sciences Laboratory of PARC, whose members primarily use another environment -- Cedar, in the great PARC tradition of developing many different programming environments.

There is a great deal of useful information available on-line at Xerox in the form of documents and source programs. Reading them is often very helpful, but finding them can be a nuisance. Throughout this document, references to on-line material are indicated by <reference>, where n is a citation reference in the bibliography at the end of this document. Standard citations to the open literature appear as [reference].

Reading a document from front to back can be mighty boring. Fortunately, this document is so disorganized that it is not at all clear that it really has a front and a back in any normal sense. You might as well just browse through and read the parts that look interesting. To help out the browsers in my reading community, I have more or less abandoned the custom of being careful to define my terms before I use them. Instead, all the relevant terms, acronyms, and the like have been collected in a separate Glossary. Some information is contained only in the Glossary, so you may want to skim through it later (or now, for that matter). While writing the Glossary, I assumed that you have a basic knowledge of computer science, and a modicum of common sense: don't expect to find terms like "computer" and "network" in the Glossary.

Naming Things

At the outset, you should know something about the names of the creatures that you will find here. The prevailing local philosophy about naming systems is perhaps somewhat different from the trend elsewhere. We do have our share of alphabet soup, that is, systems and languages that are named by acronyms of varying degrees of cuteness and artificiality; consider, for example: PARC, FTP, IFS. But we are trying to avoid making this situation any worse. To this worthy end, names for hardware and software systems are frequently taken from the Harvard Concise Dictionary of Music, or the Sunset Western Garden Book [sunset]; Grapevine servers are named after wines; Dorados are named after hotels, spices, philosphers or ships (depending on who owns them); XDE releases are named after California rivers. This convention about names does not meet with universal approval.

Local Hardware

Most of the offices and some of the alcoves around have personal computers in them of one flavor or another. The first of these was the Alto. There are more than a thousand Altos in existence now, spread throughout Xerox, the four universities in the "old" University Grant program (U. of Rochester, CMU, MIT, and Stanford), and other places. (There's a "new" University Grant program in progress.) In recent years, most of the local Altos have been replaced by various flavors of D-machines: Dorados, Dolphins, and Dandelions. Both D-machines and Altos come equipped with bitmap displays, mice, and Ethernet interfaces. Let's discuss these components first, and then turn our attention to the various personal computers that contain them.

Bitmap Displays

First, let's talk about displays. Different displas use different representations of images. A character display represents its image as a sequence of character codes. This is a very compact representation, but not a very flexible one; text is all you can get, and probably in only a limited selection of fonts. A vector display represents its image as a list of vector coordinates. This works very well for certain varieties of line drawings, but not so well for filled areas or text. A bitmap display, on the other hand, produces an image by taking a large matrix of zeros and ones, and putting white where the zeros are and black where the ones are (or vice versa). The great advantage of bitmap displays are their flexibility: you can specify a tremendous number of images by giving even a relatively small array of bits. Cursors and icons are two large classes of prominent examples. Of course, you do have to supply enough memory to hold all those bits. Altos and D-machines store their bitmaps in main storage. An alternative would be to provide a special chunk of memory on the side where the display's image sits; such a memory is often called a frame buffer.

The primary display of the Alto is a bitmap that is 608 pixels wide by 808 pixels high. Such a display is almost large enough to do a reasonable job of rendering a single 8.5" by 11" page of text. The CRT on a D-machine has the long axis horizontal instead of vertical, giving a bitmap display that is 1024 pixels wide by 808 high. It had to be 808 high so that D-machines could emulate Altos, of course. The extra space allows you to have something else on the screen as well as the somewhat scrunched page of text that you are editing.

Ere I leave you with a mistaken impression, let me note in passing that bitmap displays are not the final solution to all of the world's problems. Raster displays that can produce various levels of gray as well as black and white can depict images free of the "jaggies" and other artifacts that are inherent in bitmap displays [grayscale]. And, for some purposes, color is well worth its substantial expense.

Mice

But now on to mice. A mouse has two obvious properties--it rolls and it clicks. Inside the machine, the mouse position and the display cursor position are completely unrelated; but most software arranges for the cursor to "track" the mouse's movements. Some mice have two buttons, others three. The three mouse buttons go by various names; "left", "middle", and "right" is one set of names. The mouse buttons have at some point also been called "red", "yellow", and "blue" respectively, even though physically they are nearly always black. These colorful names were proposed at an earlier time when some of the mice had their buttons running horizontally instead of vertically. Using colors (even imaginary ones!) worked better than switching back and forth between the nomenclatures "top-middle-bottom" and "left-middle-right".

Mice also come in two basic flavors: mechanical and optical. Our current mechanical mice roll on three balls: two small ones, and one large one. Motion of the large ball is sensed by two little wipers inside the mouse, one sensing side to side rolling while the other senses forward and backward rolling. The motion of each wiper drives a commutator, and little feelers slide along the commutator, producing the electrical signals that the listening computer can decode. Building one of these little gadgets is not quite as hard as building a Swiss watch, but it's in the same league. The optical mice are a more recent innovation. An optical mouse lives on a special pad, covered with little white dots on a black background. A lens in the mouse images a portion of the pad onto the surface of a custom integrated circuit. This IC has sixteen light-sensitive regions, some of which notice that they are being shined on by the image of a white dot on the pad. As the mouse slides along the pad on its Teflon-coated underbelly, the images of the white dots move across the IC; it is subtly constructed so as to observe this phenomenon, and take appropriate electrical action. For more details on this interesting application of a custom chip, you might enjoy checking out the blue-and-white report on the subject [opticalmouse].

The Ethernet

Two's company, three's a network. A collection of machines within reasonable proximity is hooked together by an Ethernet; if that doesn't sound familiar, I know of some blue-and-whites that you might like to browse [ethernet]. Ethernets are connected to each other by Gateways and phone lines, which for most purposes allow us to ignore the topology of the resulting network. The resulting network as a whole is called an Internet. Occasionally, it's nice to know where things really are, and that's when a map <netmap> is helpful.

Ethernets come in two flavors: old and new. The old one runs at 3 MBits/sec, and should now be referred to as the t"Experimental Ethernet". The unqualified name "Ethernet" should be reserved for the new one, the standardized version used in OSD products; it runs at 10 MBits/sec.

We all know how uncommunicative computers can be when left to their own devices. That's why we invent careful protocols for them to use in talking to each other. There are two entire worlds of protocols that are spoken on our various Ethernets as well: old and new. The old ones are called PUP-based (PARC Universal Packet) [PUP]. The new ones are known by the acronym NS (Network Systems) [NS]. I'm sure that the NS protocols must be documented, but I don't know where; sorry. Each protocol world includes a hierarchy of protocols for various purposes such as transporting files, or sending and receiving mail.

In addition to connecting up all of the personal computers, the network also includes a number of machines generically called servers. Normally, servers have special purpose, expensive hardware attached to them, such as large-capacity disks, or printers. Their purpose in life is to make that hardware available to the local community. We tend to identify servers by function, so we talk about print servers, file servers, name lookup servers, mailbox servers, tape servers, and so on. Many of the protocols for use of the Ethernet were developed precisely so that personal computers could communicate effectively with servers.

The Alto

The innards of the Alto are wonderfully described in a clear and informative blue-and-white report [ALTO]. For our purposes, suffice it to say that the Alto is a 16-bit minicomputer whose primary claim to fame is that it comes equipped with a bitmap display, a mouse, and an Ethernet interface.

D-Machines

The D-machines are a family of personal computers, each member of which has a name starting with the letter "D". As long as you don't look too closely, D-machines look a lot alike. In particular, they are all 16-bit computers with a microprogrammed processor that handles most of the I/O as well running the user's programs. And they all generally come equipped with a hard disk, a bitmap display, a keyboard, a mouse, and an Ethernet interface. There are differences of course: in size, in speed, and in flexibility.

The Dolphin (formerly called the D0)

The Dolphin was one of the early D-machines, and there are still some of them around. Dolphins are housed in the same sized chassis as Altos. You can tell that they aren't Altos because they have wide screen terminals, and because they don't have a slot on top for a removable disk pack. Instead, they use a 28MByte Winchester disk drive made by Shugart. Dolphins can talk to both 3 MBit and 10 MBit Ethernets. It was sold by Xerox AI systems as the Xerox 1100 with Interlisp-D and Smalltalk.

The Dandelion

The Dandelion is the D-machine processor that is used in the Star products and the Xerox 1108. It comes in a box about half the width of an Alto chassis, and roughly the same height and depth. Dandelions are both faster and much cheaper than Dolphins. Dandelions talk only to 10 MBit Ethernets. Recently a number of hardware enhancements have been developed for Dandelions, including the ability to add a lot more memory (up to 4MByte) and hardware floating point, IBM PC Bus controller, etc.

The Dorado

The Dorado was (and probably still is) the most powerful personal computer around. The Computer Science Lab built the Dorado as the current high-performance model in the D-machine line. The processor, the instruction fetch unit, and the memory system of the Dorado have been written up in papers for your enjoyment [DORADO]. Dorados come equipped with a 315 MByte Winchester drive; older models have an 80 MByte removable-pack disk drive at present. Dorados talk only to 3 MBit Ethernets at present.

A Dorado is roughly three to five times faster than an Alto when emulating an Alto, that is, running BCPL. Dorados are generally 3-5 times faster than DLions running Interlisp, except for some notable exceptions (Dorados have an incredible memory bandwidth, and so can move the bits around the screen *really* fast; the DLions have special hardware for floating point which makes them faster than the (microcoded) Dorados.). One primary difficulty about Dorados is that they're a lot more expensive than DLions, and don't fit in your office. As a result, while some people have personal Dorados, there are also *pool* machines. When you borrow a Dorado, you generally also want to borrow at least some of the space on that Dorado's local disk. In order for this sharing to work out well, certain social taboos and customs concerning the use of such local disks have emerged, under the general rubric of "living cleanly". More on this topic anon.

In a return to the ways of the past, the Dorado processors are rack mounted in a remote, heavily air-conditioned machine room. It was initially intended that the Dorado, like the Alto, would live in your office. To prevent its noise output from driving you crazy, a very massive case was designed, complete with many pounds of sound-deadening material. (This case was known as the APC, or Armored Personnel Carrier, which it resembled.). But experience indicated that Dorados ran too hot when inside of these cabinets, and the concept of having Dorado processors in offices was abandoned. With progress in general and VLSI in particular, there is hope that the *next* high-performance contender will come out again and live in your office.

The Dicentra

The Dicentra is another D-machine of which there aren't a lot around. Essentially, it consists of the processor of the Dandelion with the tasking stuff striped out squeezed onto one Multibus card. It communicates with its memory and with I/O devices over the Multibus. Dicentras will talk to any Ethernet, or any I/O device for that matter, for which you can supply a Multibus interface card; that's one of the Dicentra's strengths. The initial application of the Dicentra is as a processor for low cost Internet gateways. The Dicentra and the Dandelion are named after wildflowers partially because they are outgrowths of an initial design called the Wildflower.

The Dragon

The Dragon is a high-performance processor based on custom integrated circuits that is being designed in CSL; confusingly enough, though, the Dragon is not really a D-machine. For example, the Dragon word size is 32 bits rather than 16. The underpinnings of Cedar will be adjusted as necessary so that Cedar will run on a Dragon; but this will take some doing.

Other D-machines

There may well be other D machines brewing in laboratories and development groups. This document won't talk about them, as they are more closely tied with some product plans that should not be widely circulated.

A few comments about Booting

All of the local processors come equipped with a hidden button called the "boot button" that is used to reinitialize the processor's state. The Alto had just one boot button, hidden behind the keyboard; pushing it booted the Alto. On Dolphins, the situation is only slightly more complex: there are two boot buttons, one at the back of the keyboard, and the other on the processor chassis itself. They perform roughly the same function, but the one on the chassis is a little more potent. On Dorados, there is a lot more going on. There are really two computers involved, the main Dorado processor and a separate microcomputer called the baseboard. It is the baseboard computer's job to monitor the power supplies and temperature and to stage-manage the complex process of powering up and down the main processor, including the correct initialization of all of its RAM's. The boot button on a Dorado is actually a way of communicating with this baseboard computer. You encode your request to the baseboard computer by pushing the boot button repeatedly: each number of pushes means something different. For details, see Ed Taft's memo on the subject <DORADOBOOT>. If the baseboard computer of the Dorado has gone west for some reason (as occasionally happens), your only hope is to push the real boot button, a little white button located on the processor chassis itself, far, far away. Just as the boot button on the keyboard is essentially a one-bit input device for the baseboard computer, the baseboard computer also has a one-bit output device: a green light located on the processor chassis. Various patterns of flashing of this light mean various things, as detailed in <DORADOBOOT>.

DLions have a boot button located right under the Maintentance Panel. It's labeled B-reset. There's another button next to it, labelled Alt-B, which is used in conjunction with B-reset to boot the machine in various ways -- you hold down both, let up on B-reset, and the maintenance panel lights cycle with numbers 0000, 0001, 0002, 0003, etc. When you let up, you get the "boot option" that you let up on: 0000 (or 0-boot) usually boots Lisp from the disk (or just B-reset), 1-boot boots Mesa from the disk, 2-boot boots from floppy, 3-boot boots from ethernet, 4-boot boots something that I forget, 5-boot gets "diagnostics" from a floppy, 6-boot gets an alternate Ethernet Mesa, and a few others. (0010-boot with a floppy-cleaning-diskette in the floppy drive will clean the floppy heads. This is useful you have a broken machine with a broken ethernet, and you want to floppy boot diagnostics but can't because of dirty heads!)

There is one more bit of folklore about booting Dorados that's fun to mention--every once in a while, I have to throw in some subtle tidbit to keep the wizards who read this from getting bored. Our subject this time is the "long push boot". Suppose that you have been working on a Dorado for a while, and you walk away to go to the bathroom. When you return and reach toward your keyboard, you get a static shock. You are only mildly annoyed at this until you notice that the cursor is no longer tracking the mouse, and the machine doesn't seem to hear any of your keystrokes. The screen looks OK, but the Dorado is ignoring all input. What has probably happened is that the microprocessor in your terminal has been knocked out by the static shock. Yes, Virginia! In addition to the Dorado itself, and the baseboard computer, there is also a microprocessor in your terminal (located in the display housing), which observes your input actions and sends them on to the main processor under a protocol referred to as "the seven-wire interface". What you want to do now is to reboot the terminal microprocessor without disturbing the state of the Dorado at all--after all, you were in the process of editing something, and you are now in danger of loosing those edits. What you should do is to depress the boot button and hold it down for quite a while (more than 2.5 seconds); and then release it. This is known as a "long push boot", and it does just what you want under these conditions: it reboots your terminal without affecting anything higher up.

Local Programming Environments

Various programming environments have grown up around the various pieces of hardware mentioned above. You can get a software merit badge simply by writing one non-trivial program in each envirnoment.

Mesa

Mesa is a strongly typed, PASCAL-like implementation language designed and built locally. It first ran on Altos. Herein, I shall call that system Alto/Mesa. Dolphins and Dorados (but not Dandelions) can run Alto/Mesa by impersonating an Alto at some level. More recent instances of Mesa now run on all of our D-machines under the Pilot operating system. In passing, I should observe that Pilot is an operating system written in Mesa by folk in SDD. It is a heavier-weight operating system than the Alto OS, providing its clients with multiprocessing, virtual memory, and mapped files.

The Pilot version of Mesa is the home to lots of active programming in several locations. First, it is the system in which the Star product was and is being implemented by OSD. The programmers in OSD have developed a set of tools for programming in Mesa variously called the "Tools Environment" or "Tajo" or "XDE" or "Basic Workstation". This body of software may soon be marketed under the name "the Xerox Development Envirnoment". In addition, Pilot Mesa is the current base of the Cedar project in CSL. More on Cedar later. Although Mesa programs look a lot like PASCAL programs when viewed in the small, Mesa although Mesa provides and enforces a modularization concept that allows large programs to be built up out of smaller pieces. The Mesa language is described by a manual [MESA].

Smalltalk

Smalltalk was developed by the folk who now call themselves the Software Concepts Laboratory (formerly known as the Learning Research Group and then the Software Concepts Group). The Smalltalk language is the purest local embodiment of "object-oriented" programming:

A computing world is composed of "objects".

The only way to manipulate an object is to be polite, and ask it to manipulate itself. One asks by sending the object a message. All computing gets done by objects sending messages to other objects.

Every object is an "instance" of some "class".

The class definition specifies the behavior of all of its instances--that is, it specifies their behavior in response to the recipt of various messages.

Genealogists will recognize that ideas from both Simula and Lisp made their way into Smalltalk, together with traces of many other languages.

For some years now, the folk in SCG have been working at trying to get the Smalltalk language and system out into the great wide world. The first public event that came out of this effort was the August 1981 issue of Byte magazine; it was devoted to Smalltalk-80, including a colorful cover drawing of the now famous Smalltalk balloon. In addition, the SCG folk are writing several books about Smalltalk, and they are planning to license the system itself to various outside vendors. The first of the books, entitled Smalltalk-80: The Language and Its Implementation, emerged from the presses at Addison-Wesley just recently [21]. Future books will include Smalltalk-80: The Interactive Programming Environment, and Smalltalk-80: Bits of History, Words of Advice.

Interlisp-D

LISP is the standard language of the Artificial Intelligence community. Pure LISP is basically a computational incarnation of the lambda calculus; but the LISP dialects in common use are richer and bigger languages than pure LISP. Interlisp is one dialect of LISP, an outgrowth of an earlier language called BBN-LISP; for more historical details, read the first few pages of the Interlisp Reference Manual [22]. One of the biggest strengths of Interlisp is the large body of software that has developed to assist people programming in Interlisp. Consider the many features of Interlisp: an interpreter, a compatible compiler, sophisticated debugging facilities, a structure-based editor, a DWIM (Do What I Mean) error correction facility, a programmer's assistant, the CLISP package for Algol-like syntax, the Masterscope static program analysis database, and the Transor LISP-to-LISP translator, to name a few.

Interlisp itself has been implemented several times. Interlisp-10 is the widely-used version that runs on PDP-10's. Interlisp-D is an implementation of Interlisp on the D-machines [23], produced by folk at PARC. In the process of building Interlisp-D, the boundary between Interlisp and the underlying virtual machine was moved downward somewhat, to minimize the dependencies of Interlisp on its software environment; that is, functions that were considered primitive in Interlisp-10 were implemented in Lisp itself in Interlisp-D. But the principal innovations of Interlisp-D are the extensions that give the Interlisp user access to the personal machine computing environment: network facilities and high-level graphics facilities (including a window package) among them.

[MORE]

Cedar

Back in 1978, folk in CSL began to consider the question of what programming environment we would use on the emerging D-machines. A working group was formed to consider the programming environments that then existed (Lisp, Mesa, and Smalltalk) and to form a catalog of programming environment capabilities, ranked by both by value and by cost. A somewhat cleaned-up version of the report of that working group is available as a blue-and-white for your perusal [EPE]. After pondering the alternatives for a while, CSL chose to build yet another programming environment, based on the Mesa language. That new environment was named "Cedar". The programming language underlying Cedar is essentially Mesa with garbage collection added. Now, adding garbage collection actually changed things quite a bit. First of all, it changes programming style in large systems tremendously. Without garbage collection, you have to enforce some set of conventions about who owns the storage. When I call you and pass you a string argument, we must agree whether I am just letting you look at my string, or I am actually turning over ownership of the string to you. If we don't see eye to eye on this point, either we will end up both owning the string (and you will aggravate me by changing my string!) or else neither of us will own it (and its storage will never be reclaimed--a storage leak). Once garbage collection is available, most of these problems go away: God, in the person of the garbage collector, owns all of the storage; it gets reclaimed when it is no longer needed, and not before. But there is a price to be paid for this convenience. The garbage collector takes time to do its work. In addition, all programmers must follow certain rules about using pointers so as not to confuse the garbage collector about what is garbage and what is not.

Thus, programs in the programming language underlying Cedar look a lot like Mesa programs, but they aren't really Mesa programs at all, on a deeper level. To avoid confusion, we decided to use the name "Cedar" to describe the Cedar programming language, as well as the environment built on top of it. Cedar is really two programming langauges: a restricted subset called the safe language, and the unrestricted full language. Programmers who stick to the safe language can rest secure in the confidence that nothing that they can write could possibly confuse the garbage collector. Their bugs will not risk bringing down the entire environment around them in a rubble of bits. Those who choose to veer outside of the safe language had better know what they are doing.

Those who want to know more about Cedar are once again encouraged to dredge up a copy of the Cedar Manual <25>. It includes documentation on how Cedar differs from Mesa, annotated examples of Cedar programs, manuals for many of Cedar's component parts, a Cedar catalog, and lots of other good stuff. By the way, the most authoritative source for what the current Cedar compiler will do on funny inputs can be found in a document called the Cedar Language Reference Manual, also known by the acronym CLRM. This is logically part of the Cedar Manual, but it is currently bound separately, and only available in draft form. The CLRM suggests a particular design philosophy for building a polymorphic language that is a superset of the current Cedar, since that is the direction in which the authors of the CLRM, Butler Lampson and Ed Satterthwaite, would like to nudge the Cedar language.

Local Software

This section is a once-over-lightly introduction to some of the major software systems that are available in the Interlisp world.

In Interlisp, the current best sources are the Interlisp Manual mentioned above.

Filing

When programming in the Alto world, or in current Cedar, you are dealing with two different types of file systems: local and remote. The local file system sits on your machine's hard disk. Remote file systems are located on file servers, machines with big disks that are willing to store files for you. Local file systems have several unpleasant characteristics in comparison with the remote systems: they are small, and they aren't very reliable. Both of these problems have consequences.

Some people believed that, because local file systems are small, it wasn't in general practical to store more than one version of a file on the local disk. Thus, some local file systems don't support versions, and in programs that use them,writing a "new version" of a file really means writing on top of the old one. Nearly everyone who isn't accustomed to this (particularly Interlisp programmers) gets burned by it at least once. Some text editors in XDE and the Alto world *do* maintain one backup copy of each file being edited as a separate file, whose name ends with a dollar sign. That is, the backup copy of "foo.bravo" is stored in the file "foo.bravo$. Note that our remote file servers do maintain multiple versions of files. Letting old versions of things accumulate is one easy way to overflow your disk usage allocation on a remote server.

No disk is completely reliable. Our remote file servers have automatic backup facilities that protect us from catastrophic disk failures. But the local file systems have no such automatic protection. Since this protection isn't provided automatically, it behooves you to adjust your behavior appropriately: make sure that, on a regular basis, backup copies of the information on your local disk are put in some safe place, such as on a remote file server where suitable precautions are constantly being taken by wizards to protect against disk failure. Doing this is one facet of what is meant by the phrase Living Cleanly, which deserves its own section.

Living Cleanly

The phrases "living cleanly" refer to a particular style of use of your local file system. In order to understand the cosmic issues involved, we should pause to discuss the ways in which local and remote file systems have been used over the years.

Back in the Alto days, personal files were usually stored on one's Alto disk pack, while project-related and other public files were stored on remote servers. Careful folk would occasionally store backup copies of their personal files on remote servers as well, in case of a head crash. But, as a general rule, one thought of one's Alto pack as the repository of one's electronic state. This made sharing Altos quite convenient, since you could turn any physical Alto into "your Alto" just by spinning up your disk pack.

In the glorious world of the future, all of your personal files as well as all public files will live on file servers in the network. The disk attached to your personal computer will, from time to time, contain copies of some of this network information, for performance reasons; but you won't have to do anything to achieve this, and you won't have to worry about how it is done. From the user's point of view, all files will act as if they were remote at all times. Indeed, except in a few funny cases, there won't even be any notion of "local file"; "file" will mean "remote file".

At the moment, we are sitting in an unpleasant trasitional phase somewhere between these two styles of usage of the local disk: we are attempting to simulate the latter state by means of manual methods and social pressure. We want you to think of your data as really living out on the file servers. That is the proper permanent home for your personal files as well as for public files. You will have to bring copies of these files, both private and public, to your local disk in order to work on them. But, at the end of each editing session, you should store the new versions of files that you have created back out to their permanent remote homes. None of this happens automatically at present; you have to make it happen manually by using various file shuffling tools, such as the "DF files" discussed below. Using these tools is a hassle, and learning how to use them can be confusing. But, there are four important benefits to be reaped from adopting a clean living life-style.

First, you are taking a step towards the glorious future.

Secondly, you are protecting yourself against failures of the local disk. A clean liver only holds information on her local disk for the duration of an editing session. This puts a reasonable bound on the amount of information that she can lose because of a disk crash.

Local file systems

The local file system in the Alto world is called either the "Alto file system" or the "BFS", the latter being an acronym for Basic File System. The biggest that a BFS can be is 22,736 pages. This is substantially bigger than the entire disk on an Alto. However, Dolphins and Dorados have much bigger local disks. Hence, when a Dolphin or Dorado is emulating an Alto, its local disk is split up into separate worlds called partitions, each containing a maximum-sized BFS. Dolphin disks can hold two full partitions, while Dorado disks can hold nineteen. What partition you are currently accessing is determined by the contents of some registers that the disk microcode uses. There is a command called "partition" in the Executive and the NetExec that allows you to change the current partition -- it necessarily "boots"the machine.

When operating in the Pilot world, a disk pack is called a physical volume, and it is divided into worlds called logical volumes.

All of our local file systems use a representation for files that drastically reduces the possibility of a hardware or software error destroying the disk’s contents. The basic idea is that you must tell the disk not only the address of the sector you want to read or write, but also what you think that sector holds. This is implemented by dividing every sector into 3 parts: a header, a label, and a data field. Each field may be independently read, written, or compared with memory during a single pass over the sector. The Alto file system stuffs a unique identification of the disk block, consisting of a file serial number and the page number within the file, into the label field. Now, when the software goes to write a sector, it typically asks the hardware to compare the label contents against data in memory, and to abort the writing of the data field if the compare fails. This makes it pretty difficult, though not impossible, to write in the wrong place. Furthermore, it distributes the structural information needed to reconstruct the file system over the whole disk, instead of localizing it in one place, the directory data structures, where a local disaster might wipe it out. Each local file system also has a utility program called a Scavenger that rebuilds the directory information by looking at all of the disk labels.

Remote file systems

The most important local file servers are IFS’s, an acronym for Interim File System (one of the crown jewels of the BCPL programming environment). Like I always say, "temporary" means "until it breaks", and "permanent" means "until we change our minds". Indigo and Ivy are two prominent local IFS’s; Indigo stores mostly project files, while Ivy stores mostly personal files. MAXC also serves as a file server for some specialized applications. Juniper was CSL’s first attempt to build a distributed transactional file server; it was one of the first large programs written in Mesa. Alpine is a new effort to build such a beast in the context of Cedar, in support of distributed databases and other such wonderful things. Some Walnut users have been storing their mail databases on Alpine for a month or more.

There is no coherent logic to the placement of "general interest" files and directories, nor even to the division between Maxc, Indigo, and Ivy. Browse through the glossary at the end of this document to get a rough idea of what’s around. If something was made available to the universities in the University Grant program, then it is probably on Maxc (or archived off of Maxc), since Maxc is the machine that the university folk can access.

IFS supplies a general sub-directory structure which the Maxc file system lacks, and as a result there are lots of place to look for a file on an IFS. For example, on Maxc you might look for

[Maxc]<AltoDocs>MyFavoritePackage.press

while on IFS you would probably look for

[Indigo]<Packages>Doc>MyFavoritePackage.press, or

[Indigo]<Packages>MyFavoritePackage>Documentation.press,

or perhaps some other permutation. This requires a bit of creativity and a little practice. However, if you get in the habit of using "*"s in file name specifications, you will find all sorts of things you might not otherwise locate. Note that a "*" in a request to an IFS will expand into all possible sequences of characters, including right angle brackets and periods. Thus, for example, a request for

<Packages>*press

refers to all files on all subdirectories of the Packages directory that end with the characters "press". A "*" won’t match a left angle bracket, by the way. Thus, if you ask for "*.press", you are referring to all Press files on the current directory. If you ask for "<*.press", you are referring to all of the Press files on the entire IFS (expect such a search to take a long time!).

There is a movement afoot in the Cedar world to simplify our file naming conventions by replacing the various flavors of brackets with a UNIX-like slash. Thus, in some Cedar systems, such as the FileTool, the documentation file mentioned above could be referred to as

/Indigo/Packages/MyFavoritePackage/Documentation.press.

File Properties

The "size" of a file is its length measured in disk pages; the "length" of a file is its length measured in bytes. The "create date" of a file is the date and time at which the information in that particular version of the file was "created", that is, the date when this that sequence of bytes came into being. Copying a file from one file system to another does not change the create date, since the information in the file, the sequence of bytes, is not affected. The create date is almost always what you want to know about a file. Some of our systems also maintain a "write date" or a "read date", but they are less well defined, and not as interesting.

Editing and Typesetting

In the outside world, document production systems are usually de-coupled from text editors. One normally takes the text that one wants to include in a document, wraps it in mysterious commands understood by a document processor, feeds it to that processor, and puzzles over the resulting jumble of characters on the page. In short, one programs in the document processor's language using conventional programming tools--an editor, a compiler, and sometimes even a debugger. Programmers tend to think this is neat; after all, one can do anything with a sufficiently powerful programming language. (Remember, Turing machines supply a sufficiently powerful programming language too.) However, document processors of this sort frequently define bizarre and semantically complex languages, and one soon discovers that all of the time goes into the edit/compile/debug cycle, not careful prose composition.

Bravo is the editor and typesetter in the Alto world, and it represented a modest step away from the programming paradigm for document production. A single program provided both the usual editing functions and a reasonable collection of formatting tools. You can't program Bravo as you would a document "compiler", but you can get very tolerable results in far less time. The secret is in the philosophy: what you see on the screen is what you get on paper. You use the editing and formatting commands to produce on the screen the page layout you want. Then, you tell Bravo to ship it to a print server and presto! You have a hardcopy version of what you saw on the screen. Sounds simple, right?

Of course, it isn't quite that easy in practice. There are dozens of subtle points having to do with fonts, margins, tabs, headings, and on and on. Bravo was a success because most of these issues are resolved more or less by fiat--someone prepared a collection of configuration parameters and a set of forms that accommodated most document production. Many of the configuration options aren't even documented, so it is hard to get enough rope to hang yourself. The net effect is that one spent more time composing and less time compiling.

In Bravo's wake, several new editors of unformatted text appeared: the Laurel editor, and the editor in the Tools Environment are prominent examples. The Laurel editor is particularly noteworthy in that it pioneered the development of a modeless (or at least less modal) user interface for an editor. The Star product editor, Tedit and Tioga are more recent local editors in the full Bravo tradition: they can handle formatting and multiple fonts. Tioga is the editor within Cedar, and its user interface is very close to the widely beloved Laurel modeless interface--try going back to Bravo after using Tioga for a while, and see how horrible it feels to have to remember to type "i" and "ESC" all the time. Tioga shows formatted text on the screen. To get a hardcopy of that text, the current path involves running a companion program called the TSetter, which will compose your pages for printing and send them to a print server. Tioga's documentation is particularly convenient, since it usually available in iconic form at the bottom of the Cedar screen <29>.

Dealing with editor bugs

All text editors have bugs. Furthermore, you are often most likely to tickle one of the remaining bugs in an editor when you are working furiously on a hard problem, and hence, have been editing for a long time without saving the intermediate results. As fate would have it, these are exactly the times when it is most damaging and most upsetting to lose your work. There is nothing quite like the sinking feeling you get when a large number of your precious keystrokes gurgle away down the drain. Both Bravo and Tioga have mechanisms that can, in some cases, save you from the horrible fate of having to do all those hours of editing over again. Bravo attempts to safeguard you by keeping track of everything that you have done during the editing session in a log file; in case of disaster, this log can be replayed to recapture most of the effects of the session. If you have a disaster when editing in Bravo, be careful NOT to respond by running Bravo again to assess the damage. By running Bravo again in the normal way, you will instantly sacrifice all chance of benefiting from the log mechanism, since the log allows replay only of the most recent session. What you want to do instead is run the program "BravoBug" ("Bravo/R" is not an adequate substitute). It wouldn't be a bad idea to ask a wizard for help also. While you are looking for a wizard, try and think of some good answer to the question "Why are you using Bravo, anyway?", which said wizard will almost certainly ask.

Printing

In general, our printers are built by taking a Xerox copier and adding electronics and a scanning laser that produce a light image to be copied. There are many different types of such printers, and there are multiple instances of each printer type as well. There are also many different programs that would like to produce printed output. The Press print file format was our first answer to the problem of allowing every printing client to use every printer. Press files are the Esperanto of printing. Most print servers demand that the documents that you send to them be in Press format. This means you have to convert whatever you have in hand (often text) to Press format before a server will deign to print it.

Press file format <PRESS> is hairy, and some print servers don't support the full generality of Press. Generally, however, such servers will simply ignore what they can't figure out, so you can safely send them any Press file you have.

A Press file can ask that text be printed in one of an extensive collection of standard fonts. Unfortunately, you must become a wizard in order to print with your own new font. You can't use a new font unless it is added to the font dictionary on your printer, and adding fonts to dictionaries is a delicate operation: a sad state of affairs. If the Press file that you send to a printer asks for a font that the printer doesn't have, it will attempt a reasonable substitution, and, in the case of Spruce, tell you about the substitution on the break page of your listing. If you have chronic font difficulties of this sort, contact a wizard.

There is a new print file format, called Interpress. The print servers that are part of the Star product speak a dialect of Interpress as will most other new Xerox printers. A print file in Interpress format is called a master. CSL's local plans for printing Interpress masters involve converting them first into a printer-dependent print file in so-called PD format (with conventional extension ".pd"). From there, a relatively simple driver program on each printer should be able to produce the final output.

PARC has a variety of printers available for your hardcopy needs. We have high volume printers for quantities of text, listings, and documentation; we have slower printers with generally higher quality for more complex files; and we have very slow printers for extremely high quality. All of our current printers except Platemaker offer 384 spots per inch and share a common font dictionary. We use two different software systems for printing Press files, both running on Altos: one is called Spruce, and the other is called (confusingly) Press. Spruce offers speed and spooling, but it can only image characters and rules, and not too many of them. This makes it limited in graphics applications. Furthermore, Spruce is limited to the particular sizes of fonts that it has stored in its font dictionary: it does not know how to build new sizes by converting from splines. Press is slower, but can handle arbitrary bitmaps, and can produce odd-sized fonts from splines.

CSL is developing Interpress printing capabilities. Printing ".pd" files is now an option on most Press printers (that is, on printers running the program Press as opposed to Spruce). Just ship your ".pd" file to the printer in the standard way: it is smart enough to figure out whether what you have sent it is in PD or Press format, and it will invoke PDPrint or Press as appropriate. Documentation on these two printing programs is available, by the way <PDPRINT, PRESS>. PD printing should not be undertaken without consultation with a wizard.

Dover printers run Spruce for high volume printing, producing a page per second. CSL's Dover, named Clover, is found in room 2106; ISL's Dover, named Menlo, is in room 2305. Samples of the Dover font dictionary may be found next to Clover and Menlo. Instructions for modifying the queue and generally running these Spruce printers are to be found next to their Alto terminals.

Lilac is CSL's color Press printer and may be found in room 2106 with Clover. It is a three color, composite-black machine; it generally produces good quality output, but is occasionally temperamental. Anyone interested in color printing or the state of Lilac should join the distribution list LilacLovers^.pa.

In room 2301, there are an assortment of black and white Press printers, answering variously to the names of RockNRoll, Quoth, and Stinger. The printers are two Ravens (Raven is a Xerox product), one Hornet, and one Gnat (the latter two are prototypes). The print quality is normally excellent. Instructions for interpreting status displays are posted locally. To be informed of which printer is functioning and where, join the list ISLPrint^.pa. There should be three printers up for most of the summer. Periodically one or another of these or Lilac are pre-empted for debugging.

Our best quality printer is Platemaker, which is normally operated at 880 spots per inch, but can be run up to 2200 spi; it is not normally useful to go beyond 1600. Platemaker uses a laser to write on photographic paper or film. Color images can be done in individual separations, which are then merged using the Chromalin process. The Platemaker printing process is used for final prints of fine images or for printing masters for publication. If you wish to have something printed, speak to Julian Orr, Eric Larson, or Gary Starkweather.

Sending and Receiving Mail

We rely very heavily on an electronic mail system. We use it for mail and also for the type of announcement that might, in other environments, be posted on a physical or electronic bulletin board. In our environment, a physical bulletin board is pretty useless, since people spend too much of their days staring at their terminals and too little wandering the halls. Electronic bulletin boards might work satisfactorily. But a bulletin board, being a shared file to which many people have write access, is a rather tricky thing in a distributed environment. It probably presupposes a distributed transactional file server, for example. Mumble. For whatever reason, the fact remains that we don't have an electronic bulletin board facility at the moment. As a result, announcements of impending meetings, "for sale" notices, and the like are all sent as messages directed at expansive distribution lists. If you don't check your messages once a day or so, you will soon find yourself out of touch (and saddled with a mailbox full of obsolete junk mail). And conversely, if you don't make moves to get on the right distribution lists early, you may miss lots of interesting mail. This business of using the message system for rapid distribution of announcements can get out of hand. One occasionally receives notices of the form: "meeting X will start in 2 minutes--all interested parties please attend".

Grapevine is the distributed transport mechanism that delivers the local mail [33]. When talking to Grapevine, individuals are referred to by a two-part name called an "R-name", which consists of a prefix and a registry separated by a dot; for example, "Ramshaw.pa" means Ramshaw of Palo Alto. In addition to delivering the mail, Grapevine also maintains a distributed database of distribution lists. A distribution list is also referred to by an R-name, whose prefix conventionally ends in the character up-arrow, as in "CSL^.pa". Distribution lists are actually special cases of a construct called a Grapevine "group". Groups can be used for such purposes as controlling access to IFS directories. There is a program named Maintain that allows you to query and update the state of the distribution list database. In fact, there are two versions of Maintain: the documented one with the unfortunate teletype-style user interface is used from within Laurel or the Mesa Development Environment <34>; the undocumented one with the futuristic menu interface is used from within Cedar. Some distribution lists are set up so that you may add or remove yourself using Maintain. If you try to add yourself to Foo^.pa and Maintain won't let you, the proper recourse is to send a message to the distribution list Owners-Foo^.pa, asking that you please be added to Foo^.

At the moment, Grapevine pretty much has a monopoly on delivering the mail. But there are several different programs that give users access to Grapevine's facilities from different environments. From an Alto, one uses Laurel, which is mentioned elsewhere as a pioneer of modeless editor interfaces. Even if you aren't a Laurel user, I recommend that you read Chapter 6 of the Laurel Manual [35], which is an enlightening and entertaining essay on proper manners in the use of the mail system. In the Mesa Development Environment, the program Hardy provides services analogous to Laurel's. From within Interlisp, most folk use Lafite, whose documentation appears as <LAFITE.PRESS>. Finally, in case travel should take you away from your multi-function personal workstation, there are servers on the Internet known by the name "Lily" to whom you can connect from any random teletype in order to peruse the mail sitting in your Grapevine mailbox.

Some Tidbits of Lore

About CSL

CSL has a weekly meeting on Wednesday afternoons called Dealer, starting at 1:15. The name comes from the concept of "dealer's choice"--the dealer sets the ground rules and topic(s) for discussion. When someone says she will "give a Dealer on X", she means that she will discuss X at some future weekly meeting, taking about 15 minutes to do so (plus whatever discussion is generated). Generally, such discussions are informal, and presentations of half-baked ideas are encouraged. The topic under discussion may be long-range, ill-formed, controversial, or all of the above. Comments from the audience are encouraged, indeed, provoked. More formal presentations occur at the Computer Forum on Thursday afternoons; the Forum is not specifically a CSL function, and it is open to all Xerox employees, and sometimes also to outsiders. Dealers are also used for announcements that are not appropriate for distribution by electronic mail. Members of CSL are expected to make a serious effort to attend Dealer.

Some Code Phrases

You may occasionally hear the following incomprehensible phrases used in discussions, sometimes accompanied by laughter. To keep you from feeling left out, we offer the following translations:

"Committing error 33"

(1) Predicating one research effort upon the success of another. (2) Allowing your own research effort to be placed on the critical path of some other project (be it a research effort or not). Known elsewhere as Forgie's principle.

"You can tell the pioneers by the arrows in their backs."

Essentially self-explanatory. Usually applied to the bold souls who attempt to use brand-new software systems, or to use older software systems in clever, novel, and therefore unanticipated ways ... with predictable consequences. Also heard with "asses" replacing "backs".

"We're having a printing discussion."

Refers to a protracted, low-level, time-consuming, generally pointless discussion of something peripherally interesting to all. Historically, printing discussions were of far greater importance than they are now. You can see why when you consider that printing was once done by carrying magnetic tapes from Maxc to a Nova that ran an XGP.

Fontology

The body of knowledge dealing with the construction and use of new fonts. It has been said that fontology recapitulates file-ogeny.

"What you see is what you get."

Used specifically in reference to the treatment of visual images by various systems, e.g., a Bravo screen display should be as close as possible to the hardcopy version of the same text. Also known is some circles by the acronym "WYSIWYG", pronuonced "whiz-ee-wig".

"Pop!"

THIS phrase means that the conversation has degenerated in some respect, often by becoming enmeshed in nitty-gritty details. Feel free to shout out one or more of these phrases if you feel that a printing discussion has been going on long enough. If two participants in a large meeting begin discussing details that are of interest to them but not of interest to the group as a whole, shout "Off-line!" instead.

"Life is hard"

Two possible interpretations: (1) "While your suggestion may have some merit, I will behave as though I hadn't heard it." (2) "While your suggestion has obvious merit, equally obvious circumstances prevent it from being seriously considered." The charm of this phrase lies precisely in this subtle but important ambiguity.

"What's a spline?"

"You have just used a term that I've heard for a year and a half, and I feel I should know, but don't. My curiosity has finally overcome my guilt." Moral: don't hesitate to ask questions, even if they seem obvious.

Hints for Gracious Living

There are a couple of areas where life at PARC can be made more pleasant if everyone is polite and thoughtful enough to go to some effort to help out. Here are a few words to the wise:

Coffee

Most groups have coffee alcoves where tea, cocoa, and several kinds of coffee are available. All coffee drinkers (not just the secretaries or some other such barbarism) help out by making coffee. If you are about to consume enough coffee that you would leave less than a full cup in the pot, it is your responsibility to make a fresh pot, following the posted instructions. There are lots of coffee fanatics around, and they get irritated beyond all reason if the coffee situation isn't working out smoothly. For those coffees for which beans are freshly ground, the local custom is to pipeline grinding and brewing. That is, you are expected to grind a cup of beans while brewing a pot of coffee from the previous load of ground beans. This speeds up the brewing process for everyone, since a load of ground beans is--at least, had better be--always ready when the coffee pot runs out.

Sharing Office Space

Be warned as well that some people are unbelievably picky about the state of their offices. The convention is that any Alto in an empty office is fair game to be borrowed. Private machines may be borrowed only by prior arrangement with their owners, because of the problems of sharing disk space. If you use someone's office for any reason, take care to put everything back exactly the way it was. Don't spill crumbs around, or leave your half-empty cocoa cup on the desk, or forget to put the machine back in the state that you found it, or whatever. Of course, lots of people wouldn't mind even if you were less than fanatically careful. But some people do mind, and there is no point in irritating people unnecessarily.

Sharing printers

When you pick up your output from a printer, it is considered antisocial merely to lift your pages off the top of the output hopper, and leave the rest there. Take a moment to sort the output into the labelled bins. Sorting output is the responsibility of everyone who prints, just as making coffee is the responsibility of everyone who drinks (coffee). Check carefully to make sure that you catch every break page: short outputs have a way of going unnoticed, and hence being missorted, especially when they come out right next to a long output in the stack. The rule for determining which bin is to use the first letter that appears in the name on the break page. Thus, "Ramshaw, Lyle" should be sorted under "R", while "Lyle Ramshaw" should be sorted under "L". A trickier question is what to do with output for "Noname", or the like. Following the rule would suggest filing such output under "N", but that doesn't seem very helpful, since the originator probably won't find it. Check the contents and file it in the right box if you happen to recognize whose output it is; otherwise, either leave it on top of the printer or stick it back in the output hopper.

The phone system

If you make a significant number of personal long-distance phone calls from Xerox phones, it is your responsibility to arrange to reimburse Xerox for them. This may not be that easy, either, since phone bills take quite a while (six weeks or so) to percolate through the bureaucracy upstairs, and the said bureaucracy also has a lot of trouble figuring out where to send the phone bills of new people, and people who move around a lot. Just because it is easy to steal phone service from Xerox doesn't make it morally right; if you think you aren't being paid enough, you should start agitating for a raise. If enough suspicious calls are made without restitution, PARC (being a bureaucracy) will impose some bureaucratic "solution" on all of us.

So as not to end on a sour note, let's discuss how the phone system works, anyway. The offices within PARC have four-digit extensions within the 494 exchange, a system known as Centrex; to dial another office, those four digits suffice. Dialing a single 9 as the first digit gives you an outside line, and you are now a normal customer of Ma Bell: see a phone book for more details (Oh, come now, surely you know about phone books!). Dialing a single 8 gives you different sounding dial tone, and puts you onto the IntelNet (not to be confused with the InterNet). The IntelNet is a Xerox-wide company phone system, complete with its own phone book, and its own phone numbers. If you are calling someone in some remote part of Xerox, you can save Mother Xerox some bread by using the IntelNet instead of going straight out over Ma Bell's lines. On the other hand, you may not get as good a circuit to talk over--although this situation is frequently said to be improving. Furthermore, through the wonders of modern electronics, you can dial any long-distance number over the IntelNet. Just use the normal area code and Ma Bell number: the circuitry is smart enough to take you as far as possible towards your destination along IntelNet wires, and then switch you over to Ma Bell lines for the rest of the trip. Using the IntelNet doesn't start to save money until the call is going a fair distance; therefore, the IntelNet doesn't let you call outside numbers in area codes 408, 415, and 916--better to just dial 9.

One more thing: after you have dialed a number on the IntelNet, you will hear a funny little beeping. At that point, you are being asked to key in a four-digit number to which the call should be billed. You should use the four-digit extension number for your normal office phone under most circumstances. Calls made by dialing 9 instead of 8 are always charged to the phone from which they are placed.

The first three rings (roughly speaking) of an incoming call occur only in your office. The next roughly three rings happen both at your office phone and at a receptionist's phone, centrally located in the laboratory. During normal business hours, the receptionist's phones are staffed; thus, someone will at least take a message for you, and leave it on a little slip of paper in your physical message box. If the second three rings go by without either of those two phones answering, the call is then forwarded to the guards desk downstairs (I believe).

If you are expecting a call but won't be near your normal phone, a call forwarding facility exists: dial 106 and then the number to which you want your calls to be forwarded. Later on (try not to forget), you dial 107 on your normal phone to cancel the forwarding. When I forward my phone, I turn the phone around physically, so that the touch-pad faces the wall. This helps me to remember to cancel the forwarding again later, at which point I turn the phone back the normal way. There is also a way to transfer incoming calls to a different Xerox number: Depress the switch hook once, and dial the destination number; when the destination answers, you will be talking to the destination but the original caller won't be able to hear your conversation; depressing the switch hook again puts all three of you on the line; then you can hang up when you please. If the destination doesn't answer, depressing the switch hook once again will flush the annoying ringing or busy signal.

References

Reference numbers in [square brackets] are for conventional hardcopy documents. Many of them are Xerox reports published in blue and white covers; the CSL blue-and-whites are available on bookshelves in the CSL Alcove. Reference numbers in <angle brackets> are for on-line documents. The path name for such files is given herein in the form

[FileServer]<Directory>SubDirectory>FileName.Extension

for backward compatibility with earlier systems. Recently, the simpler alternative form

/FileServer/Directory/SubDirectory/FileName.Extension

has begun to come into local currency, but some systems still demand brackets rather than slashes.

<n>: The generic form for a reference to an on-line document.

[n]: The generic form for a reference to a hardcopy document.

[SUNSET]: Sunset New Western Garden Book. Lane Publishing Company, Menlo Park, CA, 1979. The definitive document on Western gardening for non-botanists; 1200 plant identification drawings; comprehensive Western plant encyclopedia; zoned for all Western climates; plant selection guide in color.

[GRAYSCALE]: John E. Warnock. The Display of Characters Using Gray Level Sample Arrays. blue-and-white report CSL-80-6.

[OPTICALMOUSE]: Richard F. Lyon. The Optical Mouse, and an Architectural Methodology for Smart Digital Sensors. blue-and-white report VLSI-81-1.

[ETHERNET]: The Ethernet Local Network: Three Reports. blue-and-white report CSL-80-2.

[ETHERNET]: John F. Shoch, Yogen K. Dalal, Ronald C. Crane, and David D. Redell. Evolution of the Ethernet Local Computer Network. blue-and-white report OPD-T8102.

<TOPOLOGY>: [indigo]<AltoDocs>NetTopology.press. Contains a picture of the entire internetwork configuration in seven pages. It is out of date. All such documents are always out of date.

[PUP]: David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe. Pup: An Internetwork Architecture. blue-and-white report CSL-79-10.

[NS]: Internet Transport Protocols. Xerox System Integration Standard report XSIS 028112, December 1981.

[NS]: Courier: The Remote Procedure Call Protocol. Xerox System Integration Standard report XSIS 038112, December 1981.

[ALTO]: C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A personal computer. blue-and-white report CSL-79-11.

[DORADO]: The Dorado: A High-Performance Personal Computer; Three Papers. blue-and-white report CSL-81-1.

<DORADOBOOTING>: [Indigo]<DoradoDocs>DoradoBooting.press. Describes how to boot a Dorado, and how to configure it to boot in various ways.

[MESA]: Mesa programmers manual.

[SMALLTALK]: Adele Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. book published by Addison-Wesley, 1983.

[22]: Interlisp Reference Manual. 1983.

[23]: Papers on Interlisp-D. blue-and-white report CIS-5 (also given the number SSL-80-4), Revised version, July 1981.

[EPE]: L. Peter Deutsch and Edward A. Taft, editors. Requirements for an Experimental Programming Environment. blue-and-white report CSL-80-10.

<CEDAR>: [Indigo]<Cedar>Documentation>Manual.df. Hardcopies are entitled The Cedar Manual.

[ALTO]: Alto User's Handbook. Internal report, published in a black cover. The version of September, 1979 is identical to the version of November, 1978 except for the date on the cover and title page. Includes sections on Bravo, Laurel, FTP, Draw, Markup, and Neptune

<ALTOSYSTEMS>: [INDIGO]<AltoDocs>SubSystems.press. Documentation on individual Alto subsystems, collected in a single file. Individual systems are documented on [INDIGO]<AltoDocs>systemname.TTY, and these files are sometimes more recent than SubSystems.press.

[BRAVO]: Jerome, Suzan. Bravo Course Outline. Internal report, published in a red cover. Oriented to non-programmers.

<30>: [Indigo]<PrintingDocs>PressFormat.press. Describes the Press print file format.

<PRESS>: [Indigo]<PrintingDocs>PressOps.press. Describes the Press printing program.

<PD>: [Maxc]<PrintingDocs>PDPrintOps.press. Describes the PDPrint printing program.

[GRAPEVINE]: Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: an Exercise in Distributed Computing. blue-and-white report CSL-82-4.

<MAINTAIN>: [Ivy]<Laurel>Maintain.press. Documentation for the teletype version of Maintain, the version that is used from within Laurel or Tajo.

[LAUREL]: Douglas K. Brotz. The Laurel Manual. blue-and-white report CSL-81-6.

Â   À Á –

The Call-C-Function MISCN opcode

This opcode calls the specified C function, performaing conversion of arguments and result as needed, and returning an indication of any errors it encounters.

`(MISCN CALL-C` *Function Conversion-spec Return-Code* `&REST` *Args-To-C-Fn*`)`

Function is a Lisp integer (FIXP or SMALLP) containing the address of the function to be called. CALL-C checks for some special values, 0 (meaning the function was never loaded) and -1 (meaning the function was loaded once, and subsequently unloaded at user request), and -2 (meaning that the function has been loaded, but there are unresolved externals).

Conversion-spec specifies how the arguments and function-result are to be converted.

This is a Lisp pointer to a block of 16-bit entries:

```
+-------------------------+
+ Result Conversion Spec  |
+-------------------------+
|    Arg 0 Conversion     |
+-------------------------+
|    Arg 1 Conversion     |
+-------------------------+
|          etc.           |
+-------------------------+
|          -0-            |
+-------------------------+
```

Possible values for the conversion fields:

```
0    VOID (return only, return NIL)
1    int (Lisp SMALLP/FIXP <=> 32-bit integer)
2    char (Lisp SMALLP/CHARACTER <=> char)
3    float
4    long
5    short
6    lisp
7    cpointer
```

Return-Code is a FIXP cell into which CALL-C places a return value. Possible values are:

```
0    Successful call and return
+n   conversion error on argument n
-1   conversion error on result
-2   signal encountered while running C??
```

DEFFOREIGN—Define a foreign function for lisp.

This macro tells Medley about a foreign function—its arguments, what type of result it returns, etc. It also creates a Medley function you can call to invoke the foreign function.

`(DEFFOREIGN` *Function Result-Type ArgList* `&KEY` *:function-name*`)`

Function is a symbol, the Lisp name for the function. *Function* is given a definition that results in the foreign function being called. *Function* returns what the foreign function returns, after conversion to a Lisp datatype.

Result-Type is a symbol specifying what type of data the foreign function returns, and how it is to be converted to a Lisp type. Possible values are:

`:void`	The function returns no interesting value. Function will always return NIL.
`:long`	
`:short`	
`:int`	The function returns an integer. It is converted to a FIXP or a SMALLP.
`:char`	The function returns a character. It is converted to a SMALLP.
`:float`	The function returns a floating-point number. It is converted to a Lisp FLOATP.
`:lisp`	The function returns a lisp-pointer, which isn't converted, but DOES get reference counted.
`:byte`	?? same as character, but converted to what??
`:cpointer`	The function returns a pointer to a block of storage not in the Lisp virtual memory image. This may be a pointer to a C structure, or whatever. It is intended for use with `CBLOCKRECORD` and `DEFCSTRUCT`
<c type>	Where <C type> is a type defined using DEFCSTRUCT. The result is a pointer to a block of storage no in the Lisp virtual memory image....

ArgList is a list of symbols, each specifying what kind of data the foreign function expects for a given argument. The possible values are as above.

Function-name is a symbol or string containing the true name (as far as the foreign language is concerned) of the function you want to call when the Lisp *Function* is called.

Richard Acuff
25-Aug-84

Chat Streams

A *chat stream* is a connection between two processes oriented towards terminal service, but not necessarily restricted to that. A chat stream is inherently bi-directional so it is represented by two Interlisp-D streams; one each for input and output. The input stream is considered the primary handle on the connection and is used wherever operations are preformed that are not inherently only input or output. The following operations are available for chat streams (as well as the normal stream operations). In general these operations return true if the operation was successful, `NIL` if it could not be done:

`(`**`CHAT.SETDISPLAYTYPE`**` INSTREAM CODE)` tells the remote process that a particular type of terminal (designated by the numeric parameter `CODE`) is being emulated.

`(`**`CHAT.LOGINFO`**` INSTREAM HOST NAME)` determines if LOGIN or ATTACH should be used when logging into `HOST` with username `NAME`. This fn is only called by `CHAT` if it knows how to `ATTACH`. Returns `LOGIN, ATTACH, NIL`, or `WHERE` (when there are multiple jobs to attach to).

`(`**`CHAT.SENDSCREENPARAMS`**` INSTREAM WIDTH HEIGHT)` sends the dimentions of the virtual screen to the remote process.

`(`**`CHAT.FLUSH&WAIT`**` INSTREAM)` insures that the remote process as seen (but not necessarily acted upon) all data written so far.

`(`**`CHAT.OPTIONMENU`**` INSTREAM)` returns a `MENU` of protocol specific options that can be presented to the user.

Adding a new protocol

To add a new type of chat stream a filter function should be added to the list `CHAT.PROTOCOLS` which, when given a host name, will return a function. When this function is applied to the host name, it should return a dotted pair (`INSTREAM . OUTSTREAM`). `STREAMPROP` should be used to associate the methods for implementing the chat stream operations with `INSTREAM`. The property names are `SETDISPLAYTYPE, LOGINFO, FLUSH&WAIT, SENDSCREENPARAMS`, and `OPTIONMENU`. Only methods for implemented operations need be present.

The function (**`ADD.CHAT.MESSAGE`** `STREAM MSG`) is available for getting protocol specific messages to the user. Messages will be typed out on chat window by CHAT, or can be found on the `STREAM` property `MESSAGE` by programs. Delimiting white space is added to `MSG`.

Caveat: CHAT changes the ENDOFSTREAMOP of INSTREAM to return -1 the first time it is called, and then restore the original ENDOFSTREAMOP with the expectation that the -1 will get returned as the value of the BIN that caused the EOS op to be called. While this works for all the protocols I've tried so far, it might not work in general. This kludge is necessary in order to avoid the time-consuming alternative of looping on READP and EOFP. It also did not seem appropriate that the chat stream "interface" implement this kludge, though it might be necessary to resort to that.

Errors and related matters in CommonLoops -
A Proposal

Henry Thompson

11 August 1985

This proposal represents an attempt to provide a set of control primatives for CommonLoops which will

1) Support the existing Interlisp error handling mechanisms (including ERROR and friends, ERRORSET and friends, RESETLST and friends, ERRORTYPELST, BREAKCHECK and its consequences, and the relationships between ERRORX, FAULT1 and BREAK1, all in the context of spaghetti stacks and the existing process mechanisms;
2) Support the CommonLisp constructs catch, throw, unwindprotect, the relationship of unwindprotect to go and return(-from), error, cerror and warn;
3) Substantially reproduce the functionality of the ZetaLisp signalling facility;
4) Be a reasonably plausible attempt to take the high ground wrt whatever proposals the CommonLisp working party on error handling come up with;
5) Be a Good Thing in its own right.

Needless-to-say trying to satisfy all these goals simultaneously is not possible without some compromises, but I think what follows is a good first cut.

The starting point for this design is the Mesa signal and error mechanism, with one key idea borrowed from ZetaLisp. We start with the notion that it must be possible to unwind the stack, either as a part of non-local transfers of control, or as a consequence of abnormal termination. We add to this the notion that at certain points on the stack we may wish to take some action if such an unwinding is underway. Finally we discrimate between actions mandated at some point on the stack but taking place before the unwinding actually starts, and actions which occur at a point on the stack as the unwinding goes by.

Some terminology is in order at this point.

We call the unwinding process **unwinding**. We call the points on the stack at which action wrt unwinding may be specified *Unwind Control Points*, or *UWCPoints* for short. The process by which the user or the system announce circumstances which may provoke unwinding is called signalling, and the concrete representation of the circumstances at issue is called a condition. A condition is a CommonLoops class, and should be a sub-type of class Condition.

Unwind Control Points

UWCPoints are central to this proposal. They provide a vehicle for all activities associated with unwinding, both before the fact in th***e co***ntext of signalli*ng,* and as the stack unwinds. A UWCPoint is created with a call to the spread lambda UWCP:

`(uwcp body catch exit always-do-exits).`

Its definition is simple:

```
[prog1      (apply body nil)
      (cond (always-do-exits
             (apply exit '(normal nil))],
```

but it is what goes on behind the scenes which is important. A UWCPoint is basically a way of evaluating body (actually applying it as a function of no arguments, to allow for closures) in a context which affects what happens in the case of signalling and/or unwinding occuring within the dynamic extent of that UWCPoint. If no signalling or unwinding occurs, the value of uwcp is the value of body. If the stack is unwound past this point, `(apply exit '(unwind <condition>))` will be performed on the way past.

This suffices for unwindprotect and RESETLST (note I use `(f:l args . body)` throughout as short for `(function (lambda args . body)))`:

```
(unwindprotect form . cleanups) =>
(uwcp (f:l () form)
        nil
        (f:l (exit-key c)
           (selectq exit-key
              ((unwind normal) . cleanups)
              nil))
        t)
(RESETLST . forms) =>
(LET ((LISPXHIST LISPXHIST)
        (RESETX RESETVARSLST))
      (DECLARE (SPECVARS RESETX))
   (uwcp (f:l () . forms)
        nil
        (f:l (exit-key c)
           (selectq exit-key
              (normal (RESETRESTORE RESETX))
              (unwind (RESETRESTORE RESETX 'ERROR))
              nil))
        t))
```

RESETSAVE and RESETRESTORE are exactly as before. Note that this means that closures may appear on RESETVARSLST in case of e.g.

```
(RESETSAVE xx (LIST (f:l ...) yy))
```

but this is presumably just what is wanted.

Unwinding

The actual unrolling of the stack is performed by the spread lambda UNWIND!:

```
(unwind! frame exit-key condition).
```

frame is a stack pointer to the frame to unwind to. condition is an instance of some subclass of Condition, descriptive of what is causing the unwinding. In the simple case frame will be a UWCPoint, in which case exit-key will determine what happens when we get there.

unwind! works by scanning the stack upwards via c-links [Larry - should this be a-links? I notice that e.g. GO and RESET chase a-links, not c-links?] from its own frame until it gets to frame. Along the way, whenever it encounters a UWCPoint, it applies the exit argument of that UWCPoint to (list 'unwind condition). When it gets to frame there are two cases. If frame is a UWCPoint, then the unwinding completes with (retapply frame <the exit argument of frame> (list exit-key condition) t). If frame is not a UWCPoint, then the unwinding completes with (apply exit-key (list frame)). This latter case is for non-local 'go's and 'return(-from)'s, see Note on Non-local Xfers below.

A crucial implementation point is that unwind! releases frame before scanning the stack, and uses raw pointers during its scan. This is to prevent the stack from inadvertently being tied down if some unwind clause pre-empts the unwinding by doing its own non-local transfer, something which cannot (and indeed probably should not) be ruled out. This has the further consequence that if frame is not a UWCPoint then it must be reconstituted before being having exit-key applied to it, see above. What would make sense is for unwind! to be defined to get the raw pointer out of the stack pointer, invoke an opcode which does the actual stack scan in microcode, and then reconstruct the stack pointer and do the final apply or retapply.

Signalling

Conditions are signalled with the spread lambda raise-signal:

`(raise-signal condition can-resume neednt-catch).`

condition must be an instance of some sub-class of Condition. It identifies the circumstances which provoked the signalling, and may contain relevant parameters. If can-resume is non-nil, then the signal may be resumed, otherwise not (see below). If neednt-catch is non-nil, then the signal need not be caught, otherwise a condition Uncaught will be signalled if it is not caught.

raise-signal works by scanning the stack upwards looking for UWCPoints. When it finds one it applies the catch argument thereof to (list condition). If the result is nil, it continues the scan. If the result is non-nil we say the signal has been caught at that UWCPoint.

What happens next depends on the type of the result. If it is not a list it is called the exit key, and raise-signal exits the signal by causing the stack to unwind to the UWCPoint which caught the signal by calling (unwind! <the UWCPoint> <the exit key> condition). If it is a list then its first element is considered a resume value. If can-resume is non-nil, then the resume value is returned as the value of the call to raise-signal, otherwise an error condition CantResume will be signalled.

If the stack is scanned all the way to the top without the signal being caught, then if neednt-catch is non-nil, the value of raise-signal is nil. Otherwise, the condition Uncaught will be signalled, with instance variables recording the parameters to raise-signal. If it is exited, fine. If it is resumed, the value returned is the value of the call to raise-signal. Otherwise a break is caused around the call to raise-signal.

Note that the application of the catch argument at each UWCPoint is done in the dynamic context of the call to raise-signal, but as the catch argument is lexically in the context of the call to uwcp, if it is a closure its non-special variable references will be to that context.

Three macros are provided for the common cases:

`(signal condition) => (raise-signal condition t nil) - can resume, must be caught`

`(error condition) => (raise-signal condition nil nil) - can't resume, must be caught`

`(notify condition) => (raise-signal condition t t) - can resume, needn't be caught`

ERROR! and ERRORSET

ERROR! and control-E are now defined as (error \Abort), where \Abort is an instance of class Abort.

ERRORSET is now defined as

```
(lambda (form flag)
                (uwcp (f:l () (list (eval form)))
                      (f:l (condition)
                         (select-type condition
                            (Abort t)))
                      (f:l (exit-key condition)
                           nil))).
```

t is by convention the 'do-nothing' exit key. Note the semantics of ERRORSET are subtly changed by this definition. It is no longer the case that ERRORSET flatly stops the stack from unwinding. What it does is catch Abort, which means it short-stops ERROR!/control-E/^, as it used to, but not unwinding associated with other signals which have been caught overhead. This seems to me to be what is wanted. One could of course write a catch phrase for Condition to insure catching anything and everything, but that would be pretty dangerous.

catch and throw

These CommonLisp functions are implemented in terms of a sub-class of Condition called Throw:

```
(catch tag . body) =>
(uwcp (f:l () body)
        (f:l (condition)
           (select-type condition
              (Throw (cond ((eq condition:tag tag) 'caught)))))
        (f:l (exit-key condition)
           (selectq exit-key
              (caught condition:value)
               nil)))
(throw tag form) =>
(let ((value form))
     (raise-signal (create Throw tag_tag value_value) nil nil))
```

ERROR, ERRORX and BREAKCHECK

ERROR has a name conflict with the new (and CommonLisp) error - I propose changing its name to old-error and in the short term dicriminating on the basis of the type of the first argument. The only change to old-error is that it now passes its nobreak argument on to ERRORX, which passes it to ERRORX2. ERRORX is unchanged except for that. FAULT1 calls ERRORX2 instead of replicating it - more on this later. ERRORX2 calls BREAKCHECK as before, and then constructs an instance of class SystemError, which is a sub-class of condition, including the error number, message, position, BREAKCHK and PRINTMSG as instance variables, and signals it.

At the top of every process there is a UWCPoint, which inter alia catches Abort, and also handles most of what used to be in ERRORX2. It catches SystemError in order to implement both the built-in and user specified error type list clauses, declining to catch the signal if they don't apply. It catches Uncaught if what wasn't caught was a SystemError, and then either produces the appropriate call to BREAK1 or raises Abort, depending on the recorded value of BREAKCHK. BREAK1, however invoked, signals AboutToBreak before doing anything else. This is all a bit hairy, but the code has been worked out and will be forthcoming.

Non-Local Transfers of Control

Lexical scoping of goto tags and block labels in CommonLisp represents a bit of bother in the Interlisp context. For instance

```
(prog     ((damnFun (f:l (arg)
     (if (weird arg)
          then (go bother)
          else (process arg)))))
     (return (unwindprotect (apply* damnFun 'foo) (cleanup)))
bother
     (return 'lost))
```

works not only in the sense that if 'foo is weird, the value of the prog is 'lost, but also that in that case the unwindprotect is observed and cleanup is called. Now I don't understand how closures are to be implemented in CommonLoops, but I assume the following must be true:

1) The interpreter will continue to exist independently of the compiler (if this is false that just simplifies things a bit).

2) There is a way of identifying on the stack lexical scoping boundary points - that is to say, I presume, frames created by the application of a function definition or a non-quoted argument to apply.
3) The definition of function is such that a closure knows of every non-local lexical variable reference, goto tag and block label within it. This implies inter alia that when running interpreted all macros are expanded by function, and that evaluating or compiling calls to function may produce uba, no such tag or no such label errors.
4) The stack entry for a local variable which is referenced by a closure contains not its value but an invisible pointer to a 'free-floating' value cell, which is also pointed to by the closure.
5) When the interpreter needs the value of a lexical variable, it scans the stack up to the first boundary point and no further. The compiler will presumably be pretty much as now - collapsing all bindings upwards to the boundary frame in so far as possible, and building in references to the right points in the right frames.
6) Thus when a closure is applied to anything, a frame can be built which has entries for all its non-local lexical variable references which will do the right thing.
7) A similar, although messier, approach will work for labels and tags. Messier because although such 'free-floating' value cells may persist after their 'home' frame has gone, 'free-floating' labels and tags must be invalidated when the frame they are based on goes away (see e.g. page 41 of the CommonLisp book). Most of this hair is probably necessary simply to allow 'go's from inside nested progs in any case:

a) Compiled PROG and interpreted \PROG0 frames have two new sorts of entry for tags and labels. Each has two fields, an atom number and a pointer. Every PROG or \PROG0 frame has one tag entry with the atom number for each goto tag it owns, and one label entry for the label of the block, usually nil. In the case where no closures are involved, the pointer field of each entry corresponding to a goto tag contains the appropriate p-counter to transfer to in the case of compiled PROG frames, and the appropriate tail of the PROG body in the case of interpreted \PROG0 frames. The pointer field of the label entry contains nil. By convention (see page 120 of the CommonLisp book) every bounding frame also has a label entry for its frame-name.
b) When a go, return, or return-from is *evaluated,* the stack is scanned to the first boundary point looking for an appropriate tag/label entry. When one is found we call

```
   (unwind! <the frame> (f:l (frame) (do-go frame tag
<value of entry>)))
```

if evaluating a go, otherwise

```
(unwind! <the frame> (f:l (frame) (do-return frame label
<value of entry>
   <arg to return>)))
```

c) When a closure is constructed whose body refers to non-local tags or labels, it constructs (or finds, see below) a stack pointer for the frame in which the tag/label is bound, and includes that together with the tag/label in the closure. A pointer to this stack pointer is left in a distinguished part of the frame, so that it can be re-used by other closures, and so that it can be released when the frame goes away. Note that this means it is a special sort of stack pointer, in that its reference to the frame must not be counted.

This also implies an additional cost both in size and time to return for every frame, but I don’t see how it can be avoided.

d) When a closure is applied which includes such tags/labels, tag and label entries containing the appropriate atom numbers and the associated stack pointer are included in the constructed frame.

e) It follows from all this that do-go and do-return implement the distinction between local and non-local transfers. If the value of the entry they are passed is not a stack pointer, then they effect the local transfer, via retfrom for do-return, and by appropriate hacking of the PROG (compiled) or \PROG0 (interpreted) frame followed by retto in the case of do-go. If they do get a stack pointer, then they convert themselves into the local case by getting the entry from for the tag/label from the frame pointed to, and doing a further unwind! to that frame with an appropriate re-call of themselves as the exit-key argument. Needless-to-say, if the stack pointer has been released, we get an error.

f) All this is unnecessary for compiled transfers which don’t cross any frame boundaries, which can still be coded open.

g) It is not clear to me what will happen in the case of e.g. (eval ’(go foo)). If we take the CommonLisp manual seriously, this will fail, as it probably should, because eval will set up a bounding frame with no lexical variables or tags, but then so will most existing uses of eval... I guess this gets beyond what I can reasonably hope to second-guess...

enable

A special form is provided which will be the standard way of producing UWCPoints. It is modelled on the Cedar Mesa ENABLE form, and looks like this:

```
(enable
   c1 => a1 a2 a3 ...
     . . .
   cn => n1 n2 n3 ...
 form
   k1 -> ea1 ea2 ...
    . . .
   kn -> en1 en2 ...)
```

The double arrow lines above are called catch phrases, the single arrow lines are called exit phrases. Evaluates form so as to catch conditions c1, ... cn if they are signalled during its evaluation. If e.g. c1 is signalled, the forms a1 ... an (the catch phrase for c1) will be evaluated in the context of the call which signalled c1. Catch phrases are implemented with select-type, so the order of the condition names is significant. For a catch phrase to be well formed, all control paths through it must end with one of the following four quit forms:

```
(exit)
```

Causes the stack to unwind back through the enclosing enable form, which is exited with value NIL.

```
(resume form)
```

Returns from the call which did the signalling with the value of form as the value of that call, if it is resumable, otherwise generates an error.

```
(goto exit-key)
```

Causes the stack to unwind back to the enclosing enable form, where the exit phrase for exit-key is evaluated. The value of the last form in the phrase is the value of the enable. Exit phrases are implemented with SELECTQ, so lists of exit keys may precede ->.

```
(reject)
```

Causes the signal handling process to act as if the catch phrase had not been there at all.

There is a special exit key whose name is unwind, which has a special meaning. The exit clause for the unwind key will be evaluated whenever the stack unwinds upwards past this point.

There is another special exit key whose name is normal. Its exit clause will be executed in case of a normal return from the enclosed form, without affecting the value of the enable, which will still be the value of the enclosed form.

There is another special exit key whose name is always. always is a just a synonym for (normal unwind). Thus its exit clause will be executed if the stack ever unwinds past it and it will also be executed in case of a normal return from the enclosed form.

Calls to enable translate into calls to uwcp as follows, taking the above template for enable as the input:

```
(uwcp (f:l () form)
          (f:l (condition)
             (select-type condition
                   (c1 a1 a2 a3 ...)
                      . . .
                   (cn n1 n2 n3 ...)))
          (f:l (exit-key condition)
             (selectq exit-key
                (k1 ea1 ea2 ...)
                   . . .
                (kn en1 en2 ...)
                (t nil)
                nil))
          <if normal or always appeared then t else nil>)
```

```
(exit) =>
t
```

```
(resume form) =>
(list form)
```

```
(goto exit-key) =>
(quote exit-key)
```

```
(reject) =>
nil
```

Built in Condition classes

Here follow the definitions of class Condition and its built in sub-classes:

```
(defstruct Condition "an unspecialised condition")
```

[Note: It may turn out to be useful to include here instance variables can-resume and neednt-catch which are used instead of the arguments to raise-signal.]

```
(defstruct (Abort (:include Condition)) "an abort condition")
```

```
(defstruct (Throw (:include Condition)) "a condition for throwing
to a catch"
   tag value)
```

```
(defstruct (SystemError (:include Condition))
   "a condition resulting from a call to ERRORX"
   number message stack-pos breakchk printmsg)
```

```
(defstruct (Uncaught (:include Condition))
```

```
   "a condition resulting from an uncaught signal" condition)
(defstruct (AboutToBreak (:include Condition))
   "a condition signalled by BREAK1 on entry")
```

Relation to CommonLisp

This proposal supports everything relevant in the CommonLisp book.

Interlisp to Common Lisp Concordia

Chapter 2 IRM (Datatypes)

Interlisp Form	Common Lisp Form
(DATATYPES --)	??
(TYPENAME datum)	?? (type-of datum) -- except for strings and arrays Note that the result types are different, however, and it is necessary to check for literals in the program, e.g., (IL:TYPENAME 123) => IL:SMALLP yet (IL:TYPE-OF 123) => LISP:FIXNUM. Also LISP:TYPE-OF is definitely non-portable except for structures.
(TYPENAMEP datum typename)	?? (typep datum typename) -- except for strings and arrays and the problem of non-portability of the type names.

2.1 Datatype Predicates

**For many of these, the translation should look at the value/effect context.
If used for effect only, no need to insert the (and (<test> x) x).**

(LITATOM x)	(symbolp x)
(SMALLP x)	(and (typep x ’fixnum) x)
(FIXP x)	(and (integerp x) x)
(FLOATP x)	(and (floatp x) x)
(NUMBERP x)	(and (numberp x) x) -- but includes more sorts of numbers
(ATOM x)	(and (or (symbolp x) (numberp x)) x) Often users wrote IL:ATOM when they meant the LISP:ATOM interpretation, however.
(LISTP x)	(and (consp x) x)
(NLISTP x)	(not (consp x)) or (atom x)
(STRINGP x)	(and (stringp x) x)
(ARRAYP x)	?? How are arrays to be represented? possibly (and (vectorp x) x) BVM - "ARRAYP probably translates as vectorp. Again, the real question is how ARRAY translates, at least when the origin is 1 (the default). You could translate to make-array with a size one greater than specified (wasting the zero element), but then you can’t translate ARRAYSIZE as length. Sigh."
(HARRYP x)	(and (hash-table-p x) x) -- Not quite strong enough since Interlisp hash tables are more general than CL ones BVM -- "hash-table-p is probably good enough; it’s the translation of HASH-ARRAY that will need more strength."

2.2 Datatype Equality

(EQ x y)	(eq x y)
(NEQ x y)	(not (eq x y))
(NULL x)	(null x)
(NOT x)	(not x)
(EQP x y)	(or (eq x y) (and (numberp x) (number y) (= x y))) Probably (= x y) will suffice in most cases BVM - "EQP also compares compiled code, but there's not much hope there."
(EQUAL x y)	?? Probably (equal x y) will suffice in most cases (differ on number comparisons and the CL version descends more datatypes)
(EQUALALL x y)	?? Probably (equalp x y) will suffice in most cases (differ on string comparisons)

2.3 Fast and Destructive Functions

2.4.1 Using Litatoms as Variables

(BOUNDP var)	(boundp var)
(SET x y)	(set x y) Note that this is a place where free variable references might "sneak" in and ruin the automatic "only declare special things that are used free." algorithm.
(SETQ x y)	(setq x y)
(SETQQ x y)	(setq x 'y)
(GETTOPVAL var)	?? (symbol-value atom) -- no concept to top level value in CL BVM -- "I would translate GETTOPVAL and SETTOPVAL as symbol-value and set (not identity and setq), with a warning that they're wrong."
(SETTOPVAL var value)	?? (set var value)
(GETATOMVAL atom)	(symbol-value atom) BVM - "{GET|SET}ATOMVAL are exactly symbol-value and set, with the implicit declaration, irrelevant to common lisp, that the variable is not dynamically bound."
(SETATOMVAL atom value)	(set var value)

2.4.3 Property Lists

(GETPROP atom prop)	(get atom prop) BVM -- "GETPROP is really (and (symbolp atom) (get atom prop)), though you'll usually want it translated directly as get. Fortunately, PUTPROP does not suffer this brain damage."
(PUTPROP atom prop val)	(setf (get atom prop) val)

(ADDPROP atom prop new flg)	?? -- no direction translation (runtime?)
(REMPROP atom prop)	(remprop atom prop)
(REMPROPLIST atom prop)	?? -- no direction translation (runtime?)
(CHANGEPROP x prop1 prop2)	?? -- no direction translation (runtime?)
(PROPNAMES atom)	?? -- no direction translation (runtime?)
(DEFLIST l prop)	?? -- no direction translation (runtime?) LMM -- "Surely obsolete and not necessary."
(GETPROPLIST atom)	(symbol-plist atom)
(SETPROPLIST atom list)	(setf (symbol-plist atom) list)
(GETLIS x props)	??

```
(multiple-value-bind (prop value tail)
        (get-properties (symbol-plist x) props)
                tail)
```

2.4.4 Print Names

Most of this section is extremely problematic -- especially since, although functions may be written that capture much of the semantic content, they tend to much more cons'y then their Interlisp counterparts, hence will disrupt the performance profile of any translated program that exploits these features.

AD -- "I'd be tempted to leave most of the atom-building functions untranslated and flag them as something that the programmer should deal with himself. Except for very simple things, you will probably want to do whatever you were doing with atoms in some other way in CL."

BVM -- "I tend to agree with Andy. However, some of these are common enough that it might be worthwhile having approximate definitions in the library. E.g., write a version of MKATOM that does ordinary strings and numbers (the definition I wrote is close; slightly better might be one that did read-from-string while binding *readtable* to a table in which all the special characters have been given alphabetic syntax). Translate SUBATOM, PACK, PACK* as (MKATOM something), and then just flag all the MKATOMs uniformly.

It doesn't seem worth even trying for UNPACK, as any use is highly likely to need manual intervention anyway."

BVM - "Given that IL is so willing, and CL so unwilling, to coerce to strings, you might introduce a coerce-to-string macro to make some "translations" more palatable. If the translator knows how to evaluate it for constant forms (such as strings), so much the better."

(MKATOM x) — ?? This is hard to capture exactly -- but here's one attempt

```
(defun mkatom (arg)
        (if (numberp arg)
                arg
                (values
                  (intern
                        (typecase arg
                          (symbol (string arg))
                          (string arg)
                          (otherwise
                                (prin1-to-string arg)))
                        ))
))
```

```
and another (due to BVM)
(defun mkatom (arg)
  (let ((string (typecase arg
                    (symbol (string arg))
                    (string arg)
                    (otherwise (prin1-to-string arg)))))
        (multiple-value-bind (n end)
                    (parse-integer string :junk-allowed t)
                      (if (and n (= end (length string)))
                            n
                            (values (intern string
                                        ))))
)) BVM -- "Of course, this still doesn't do (mkatom "123Q") or
(mkatom "12E3") correctly (yecch)."
```

(SUBATOM x n m)

```
??
Again here's a (long and cons'y) attempt at translation
(defun subatom (x n &optional  (m -1))
(let* ((string (symbol-name x))
        (start (if (< n 0)
                    (+ (length string) n)
                    (1- n)))
        (end (if (< m 0)
                    (+ 1 (length string) m)
                    m)))
        (values (intern (subseq string start end)))
))
or
(MKATOM (subseq
                (string x)
                (if (< n 0 )
                  (+ (length string) n)
                  (1- n))
                (if (< m 0 )
                  (+1 (length string) m)
                  m)))
```

(PACK x)

```
??
But try
(defun pack (arglist)
  (let ((new-arglist
          (mapcar
              #'(lambda (arg)
                  (typecase arg
                      (symbol (string arg))
                      (string arg)
                      (otherwise
                       (prin1-to-string arg))))
              arglist)))
   (values (intern
        (apply #'concatenate 'string new-arglist)))
))
or
(MKATOM
(apply
  #'concatenate
  'string
  (mapcar
        #'(lambda (arg)
              (typecase arg
                  (symbol (string arg))
                  (string arg)
                  (otherwise
```

```
                                       (prin1-to-string arg))))
    x)))
```

(PACK* x1 x2 .. xn)

??
But try

```
(values
  (intern (apply #'concatenate
                 'string
                 (mapcar #'princ-to-string
                         (list x1 x2 .. xn)))))
```

or

```
(MKATOM
  (apply
      #'concatenate
      'string
      (mapcar #'princ-to-string
              (list x1 x2 ".." xn))))
```

(UNPACK x flg rdtbl)

??
But try

```
(defun unpack (arg)
  (let ((string (typecase arg
                  (symbol (string arg))
                  (string arg)
                  (otherwise (prin1-to-string arg))))
        (result nil)
        (ch nil))
    (with-collection
      (dotimes (i (length string))
        (setq ch (char string i))
        (collect
                  (or (digit-char-p ch)
                      (intern
                           (string (char string i))))
        )))
))
```

A more Common Lisp'y version is:

```
(defun unpack (arg)
(let ((string
        (typecase arg
                  (symbol (string arg))
                  (string arg)
                  (otherwise (prin1-to-string arg))))
      )
(coerce string 'list)          ))
```

(DUNPACK x scatchlist flg rdtbl)

"
BVM -- "I see no need for DCHCON and DUNPACK to translate differently than CHCON and UNPACK, though the translations may want to be flagged (but then, you need to flag them anyway)"

(NCHARS x flg rdtbl)

??

```
(defun nchars
        (arg &optional (flg nil)
                  (*readtable* *readtable*))
        (length
                  (if flg
                      (prin1-to-string arg)
                  (princ-to-string arg)))
)
```

If flg is nil, this can be optimized to cut down on the consing.

(NTHCHAR x n flg rdtbl)

```
??
(let ((*readtable* (or rdtbl *readtable*)))
  (if flg
        (values (intern
                      (aref (prin1-to-string x) (1- n))))
        (values (intern
                      (aref (princ-to-string x) (1- n))))
)
```

Use of this function almost surely indicates a stylistic problem -- single letter symbols being used as character objects

(L-CASE x flg)

```
??
(typecase x
        (string (if flg
                      (string-capitalize x) ;;not quite
                      (string-downcase x)))
        (symbol
              (values (intern
                          (if flg
                               (string-capitalize x)
                               (string-downcase x)))))
        (cons
              (mapcar #'L-CASE x)))
```

(U-CASE x)

```
??
(typecase x
        (string (string-upcase x))
        (symbol
              (values (intern
                          (string-upcase x))))
        (cons
              (mapcar #'U-CASE x)))
```

(GENSYM char)

```
(gensym (if char (string char)))
```

Although this translation may well in subtle ways

GENNUM

?? -- no corresponding var in Common Lisp

(MAPATOMS fn)

```
(do-all-symbols (dummy-var)
        (funcall fn dummy-var))
```

Although do-all-symbols is not guaranteed to touch each symbol only once.

2.4.5 Character Code Functions

This section forces to face squarely the problem of Interlisp's penchant of representing character objects as symbols with single letter p-names.

(PACKC x)

```
??
(MKATOM (coerce
                (mapcar #'code-char x) 'string))
```

(CHCON x flg rdtbl)

```
??
(mapcar #'(lambda (sym)
                (char (symbol-name sym) 0))
        (UNPACK x flg rdtbl))
```

(DCHCON x scatchlist flg rdtbl)

"

(NTHCHARCODE x n flg rdtbl)

```
??
(char-code (char (symbol-name
                (NTHCHAR x n flg rdtbl)) 0))
```

Not quite right since NTHCHARCODE may return NIL in some circumstances

(CHCON1 x)

??
(char-code (char (symbol-name x) 0))
BVM - "Your translation of CHCON1 oddly assumes the arg is a symbol, rather than an arbitrary printable object"

(CHARACTER n)

??
(MKATOM (string (code-char x)))

(FCHARACTER n)

"

(CHARCODE c)

??
```
(defun charcode-1 (c)
   (etypecase c
            (symbol
                     (case symbol
                       (CR 13)
                       ...
                       (otherwise
                         (char-code (char
                                            (symbol-name c)
                                            0)))))
            (string
                      (char-code (char c 0)))
            (cons
                     (cons (charcode-1 (car c))
                          (charcode-1 (cdr c)))))
)
(defmacro charcode (c)
            (charcode-1 c ))
```

or in many cases
(char-code "some character object")
BVM - "CHARCODE should probably *always* translate as (char-code #\somechar), to facilitate conversion to the character idiom."

(SELCHARQ e c1 .. cn default)

```
(defmacro (e &rest args)
   (let ((default (car (last args)))
           (clauses (butlast args 1)))
     ‘(SELECTQ ,e
            ,@(mapcar
                     #’(lambda (clause)
                              ‘(, (CHARCODE (car clause)) .
                                       ,@(cdr clause))) clauses)
            ,default))
)
```

2.5 Lists

(CONS x y)	(cons x y)
(CAR x)	(car x)
(CDR x)	(cdr x)
(CAAR x)	(caar x)
.....	.....
(CDDDR x)	(cdddr x)
(RPLACD x y)	(rplacd x y)
(FRPLACD x y)	"

(RPLACA x y)
(FRPLACA x y)

```
(rplaca x y)
"
```

(RPLNODE x a d)
(FRPLNODE x a d)

```
(rplacd (rplaca x a) d)
"
```

(RPLNODE2 x y)
(FRPLNODE2 x y)

```
(rplacd (rplaca x (car y)) (cdr y))
"
```

2.5.1 Creating Lists

(MKLIST x)

```
(if (listp x) x (list x))
```

(LIST x1 x2 .. xn)

```
(list x1 x2 .. xn)
```

(APPEND x1 x2 .. xn)

```
(append x1 x2 .. xn)
```

(APPEND x)

```
(copy-list x)
```

(NCONC x1 x2 .. xn)

```
(nconc x1 x2 .. xn)
```

(NCONC1 lst x)

```
(nconc lst (list x))
```

(ATTACH x l)

```
?? -- probably obsolete
(defun attach (x l)
  (if (null l)
        (cons x l)
        (progn (setf (cdr l)
                      (cons (car l) (cdr l)))
               (rplaca l x)))
)
```

2.5.2 Building Lists from Left to Right

(TCONC ptr x)

```
??
(defun tconc (ptr x)
  (let ((head (car ptr))
        (tail (cdr ptr)))
    (if (null head)
        (let ((result (list x)))
          (cons result result))
        (progn (setf (cdr ptr)
                     (cdr (rplacd tail (list x))))
               ptr)))
)
```

(LCONC ptr x)

```
??
(defun lconc (ptr x)
  (let ((head (car ptr))
        (tail (cdr ptr)))
    (if (null head)
        (cons x (last x))
        (progn (setf (cdr ptr)
                     (last (rplacd tail x)))
               ptr)))
)
```

(DOCOLLECT item lst)

```
??
```

(ENDCOLLECT item tail)

```
??
```

2.5.3 Copying Lists

(COPY x)	(copy-tree x)
(COPYALL x)	??
(HCOPYALL x)	??

Note from LMM:

"I've no trouble with your LIST translations. Are you sure CL has RPLACD? I thought you have to do (progn (SETF (CDR x) y) x).

I think the Interlisp character functions point up a kind of design choice that will come up again and again, in situations where the fundamental mechanism for getting something done in CL and IL differ.

I think a the translator might offer three choices:

(a) leave the functions alone (e.g., translate to IL:DCHCON and IL:MKATOM which are defined in a "compatibility" package). This gives code that works.

(b) produce "interim" translations, which have the same effect, e.g., as you've identified in your last message.

(c) attempt to produce "natural" Common Lisp style (examples follow.)

In the case of an Interlisp program that does PACKC, CHCON, DCHCON, in some cases the "native" CL program would use strings, and others, it would use symbols. (Interlisp programmers use symbols where CL programmers would use strings.)

Usually, the "native" trnaslation of CL functions that deal in character codes is to translate them to deal in character objects. Sometimes, where an IL programmer deals with a list of characters or character codes, the CL programmer would leave it as a string; the problem was that IL didn't have the breadth of sequence functions and so IL programmers would frequently hack lists.

If IL:character/code/atom/list == CL: character/character/string/list

then

(PACKC x) => (coerce 'string x)
(CHCON x) => (coerce 'list x)
(CHCON x flg rdtbl) => (coerce 'list (write-to-string x))

Ignore & flag RDTBL argument

(DCHCON ...) => ignore & use CHCON

(NTHCHARCODE ...) => SCHAR

CHCON1 => SCHAR ... 1

CHARACTER => no-op
FCHARACTER

CHARCODE => use #\.

SELCHARQ => CASE with #\ as case elements"

2.5.4 Extracting Tails of Lists

(TAILP x y)
(tailp x y)

(NTH x n)
((lambda (list index) (nthcdr (1- index) list)) x n)
BVM - "NTH returns tails, is one-based and has stupid behavior for n < 1"

(FNTH x n)
"

(LAST x)
(last x)
Although the behavior of last on non-list is not defined

(FLAST x)
"

(NLEFT l n tail)
??

```
(defun nleft (l n tail)
  (if (and tail (tailp tail l))
        (let* ((length (length l))
               (sub-length (length tail))
               (diff (- length sub-length n)))
          (if (>= diff 0)
                (dotimes (i diff l)
                      (setq l (cdr l))))))
)
```

BVM - "The CL translation of the Interlisp definition of NLEFT would be substantially better than the one you give."

(LASTN l n)
?? is LASTN destructive?

2.5.5 Counting List Cells

EQLENGTH, COUNTDOWN, and EQUALN are applicable to circular lists.
BVM - "I think worrying about il:equal is a waste of energy. The subtle difference between il:equal and cl:equal should be globally noted as a potential, albeit unlikely, source of incompatibility."

(LENGTH x)
(length x)
Although length is only defined for true lists

(FLENGTH x)
"

(EQLENGTH x n)
(eql (length x) n)
Although would fail to return if x were circular
BVM - "For its non-circularity consideration, a more faithful translation might be ((lambda (tail) (and (consp tail) (atom (cdr tail)))) (nthcdr (1- n) x)), but it is less obvious what is going on."

(COUNT x)
??

```
(defun count (x)
  (+ (length x)
     (let ((sum 0))
        (dolist (a x)
          (if (consp a)
                (incf sum (count a))))))
)
```

(COUNTDOWN x n)
??

(EQUALN x y depth)
??

```
(defun equaln (x y depth)
```

```
(cond ((eq depth 0) t)
      ((consp x)
          (and (consp y)
               (equaln (car x) (car y) (1- depth))))
      (t
          (and (not (consp y))
               (equal x y))))
)
```

NB equal not equivalent to IL:EQUAL

2.5.6 Logical Operators

(LDIFF x y)	(ldiff x y) Except if y is not a tail of x. (LDIFF would signal an error in this case while ldiff would return (copy-list x)) NB -- if y is nil (LDIFF x y) -> x BVM - "You might want to recognize the idiom (LDIFF lst (NLEFT lst n)) as (butlast lst n)"
(LDIFFERENCE x y)	(set-difference x y :test #'equal) NB. equal is not equivalent to IL:EQUAL.
(INTERSECTION x y)	(intersection x y :test #'equal) NB. equal is not equivalent to IL:EQUAL. Elimination of duplicate entries is not guaranteed by CLtL. BVM - "The fact that INTERSECTION advertises duplicate removal suggests that the conservative translation should be (remove-duplicates (intersection x y :test #'equal) :test #'equal)" BVM - "Recognize the common idiom (INTERSECTION x x) as (remove-duplicates x :test #'equal)"
(UNION x y)	(union x y :test #'equal) NB. equal is not equivalent to IL:EQUAL. Again -- treatment of duplicate entries may not be identical.

2.5.7 Searching Lists

(MEMB x y)	(member x y :test #'eq) Not defined if y is not a true list
(FMEMB x y)	"
(MEMBER x y)	(member x y :test #'equal) NB. equal is not equivalent to IL:EQUAL
(EQMEMB x y)	(or (eq x y) (and (consp y) (member x y :test #'eq)))

2.5.8 Substitution Functions

(SUBST new old expr)	(subst new old expr :test #'equal) NB. equal is not equivalent to IL:EQUAL. With this translation, if new is a consp, then new will NOT be copied on each substitution.
(DSUBST new old expr)	(nsubst new old expr :test #'equal) Same caveat as for SUBST
(LSUBST new old expr)	?? (sort of an nconc subst)

(SUBLIS alst expr flg)

```
(if (null flg)
    (sublis alst expr :test #'equal)
   ??)
```

NB. The usual equal caveat holds. If (eq flg t) then SUBLIS is required to cons an entirely new tree

(DSUBLIS alst expr flg)

```
(if (null flg)
   (nsublis alst expr :test #'equal)
 ??)
```

Same caveat as SUBLIS

(SUBPAIR old new expr flg)

??
ignoring flg and the strange behavior with respect to non-nil final tails of old, roughly equivalent to:
(sublis (mapcar #'cons old new) expr :test #'equal)

2.5.9 Association Lists and Property Lists

(ASSOC key alst)

(assoc key alst :test #'eq)
But not equivalent if alst is not a true list

BVM - "For ASSOC, DREMOVE, etc, I think you should use test eql instead of eq (in fact, isn't that the default in cl?). This is actually a more widespread and difficult problem with translating IL code--the hidden assumption that a substantial class of integers are immediate and hence testable by eq."

JOP - "I'm not sure I agree with the rational for using eql rather than eq in ASSOC (and friends), for the following reasons: (a) the keys for ASSOC (etc.) are usually symbols, and (b) Although not explicitly stated in CLtL -- it's probably fairly safe to assume that eq comparisons are valid for fixnums."

BVM - "I thought CLtL did explicitly state (p. 193) that it is NOT safe to assume that eq comparisons are valid for fixnums. This is not to say that I am aware of any implementations in which eql fixnums are not eq. However, there are certainly implementations in which the fixnum range is considerably smaller than ours, another subtle obstacle in porting. As for your point (a), it is my impression that people are fairly sloppy about whether assoc keys are symbols or not. Aside from all that, there's a reason that CLtL's default for assoc, etc, is eql. I think that translating il:assoc directly as cl:assoc is appropriate style; at worst, it performs slightly less efficiently than with an eq test, but you know it won't be wrong."

LMM - "The decision of EQ vs EQL in ASSOC is probably one of those decisions to made interactively at translation time..."

(FASSOC key alst)

"

(SASSOC key alst)

(assoc key alst :test #'equal)
Usual caveat about equal -- non NIL tails of alst

(PUTASSOC key val alst)

??

```
(defun putassoc (key val alst)
   (let ((entry (assoc key alst :test #'eq)))
      (if entry
          (setf (cdr entry) val)
      (progn (nconc alst (cons key val))
                    val)))
)
```

	LMM - "I've found on more than one occasion that to do a "natural" translation, I wound up changing an ALIST into a property list, e.g., so I could use (SETF (GETF x y) z) instead of (PUTASSOC x y z)."
(LISTGET lst prop)	(getf lst prop)
(LISTPUT lst prop val)	(setf (getf lst prop) val)
(LISTGET1 lst prop)	(cdr (member prop lst :test #'eq)) NB. Order of evaluation not preserved
(LISTPUT1 lst prop val)	(setf (cdr (member prop lst :test #'eq)) val) NB. Order of evaluation not preserved

2.5.10 Other List Functions

(REMOVE x l)	(remove x l :test #'equal) NB. equal not equivalent to IL:EQUAL
(DREMOVE x l)	(delete x l :test #'eq) Not guaranteed to return an eq list if the result is non-nil
(REVERSE l)	(reverse l) Not equivalent if l is not a list
(DREVERSE l)	(nreverse l) Same caveat as REVERSE

2.6 Strings

Some thorny issues arise here. Among them: (a.) Some Interlisp string functions will clearly not be applicable to all types of strings (eg GNC GLC), (b.) Some agreement must be attained between the allowable set of character objects and string-chars -- this may confine us to the 96 standard characters, excluding control characters, NS characters, etc. (c.) Reusing string headers is a fairly inoperative idea -- although doable if the reusable string is adjustable (d.) It may be nice to have some general technology for mapping from a index-origin-one indexing scheme to a index-origin-zero indexing scheme. This may include fairly global source modifications

(STREQUAL x y)	(string= x y)
(ALLOCSTRING n initchar)	(make-string n :initial-element (char-code initchar))
(ALLOCSTRING n initchar old)	?? (adjust-array old n :initial-element (char-code initchar))

(MKSTRING x flg rdtbl)

??

```
(defun mkstring
     (x &optional flg (rdtbl *readtable*))
  (let ((*readtable* rdtbl))
     (if (null flg)
            (typecase x
              (string x)
              (symbol (symbol-name x))
              (otherwise
                 (princ-to-string x)))
     (prin1-to-string x))
))
```

(SUBSTRING x n m)

??

```
(defun substring (x n &optional (m -1))
  (let* ((length (length x))
```

```
              (start (if (< n 0)
                              (+ length n)
                             (1- n)))
              (end (if (< m 0)
                              (+1 length m)
                             m)))
    (make-array (- end start)
              :element-type 'string-char
              :displaced-to x
              :displaced-index-offset start)
))
```

(SUBSTRING x n m oldptr)

??
Might be able to do something if oldptr were an adjustable string

(GNC x)

??
Requires x to be adjustable

```
(defun gnc (x)
   (let ((holder (make-array (length x)
                                      :element-type 'string-char
                                      :displaced-to x)))
     (prog1 (char x 0)
            (adjust-array x (1- (length x))
                          :displaced-to holder
                          :displaced-index-offset 1)))
)
```

I'm not sure what would happen if the translation were simply

```
(prog1 (char x 0)
            (adjust-array x (1- (length x))
                          :displaced-to x
                          :displaced-index-offset 1))
```

Note that a character object is returned rather than a symbol

(GLC x)

??
x required to have a fill-pointer

```
(vector-pop x)
```

Note that a character object is returned rather than a symbol

(CONCAT x1 x2 .. xn)

??

```
(concatenate 'string (MKSTRING x1)
                              (MKSTRING x2)
                              ..
                              (MKSTRING xn))
```

(CONCAT)

```
(make-string 0)
```

(CONCATLIST x)

```
(apply #'CONCAT x)
```

(RPLSTRING x n y)

??

```
(defun rplstring (x n y)
   (let ((start (if (< n 0)
                           (+ (length x) n)
                          (1- n))))
    (do ((i 0 (1+ i))
            (limit (length y))
            (j start (1+ j)))
           ((eql j limit) x)
      (setf (char x j) (char y i))))
)
```

(RPLCHARCODE x n charcode)

??

```
(defun rplcharcode (x n charcode)
   (let ((index (if (< n 0)
```

```
                    (+ (length x) n)
                   (1- n))))
 (Setf (char x index) (char-code charcode))
 x)
)
```

(STRPOS pat string start)	?? roughly (1+ (search pat string :start1 (1- start))
(STRPOS pat string start skip anchor tail)	??
(STRPOSL a str strat)	?? roughly (1+ (search (mapcar #'code-char a) str :start1 (1- start)))
(MAKEBITTABLE l neg a)	??

2.7 Arrays

Suppose Interlisp arrays are represented by Common Lisp vectors, then two strategies present themselves for translation of the array facilities.
a.) Perform everywhere suitable subtractions -- but attempt global code simplification
b.) Use an addition vector cell and preserve origin-1 indexing
I will attempt to list translations appropriate for both strategies

NB. The index origin of a translated Interlisp vector will not be knowable at run-time.

BVM - "Since you can't tell by looking at a call to ELT or SETA whether the array is 0- or 1-origin, you can only use method "a" (subtract 1 from all indices) if the user is willing to globally declare "I never use zero-origin arrays". Note that when using method "b", you have to inflate the size of the vector by 1 even on calls to ARRAY with origin constant zero, unless you never care about ARRAYSIZE translating correctly."

Interlisp array element-types may be translated as follows

BIT	bit
BYTE	(unsigned-byte 8)
WORD	(unsigned-byte 16)
FLOATP	float
POINTER	t
DOUBLEPOINTER	??
XPOINTER	??
FLAG	(member t nil)
(BITS n)	(unsigned-byte n)
FIXP	(signed-byte 32) or t
SIGNEDWORD	(signed-byte 16)

One might imagine two functions -- translate-type and inverse-translate-type -- to move from Interlisp types to Common Lisp types and back again

(ARRAY size type init)	a.) (make-array size :element-type (translate-type type) :initial-element init) b.) (make-array (1+ size) :element-type (translate-type type) :initial-element init) Of course, if the array origin is explicitly specified as zero (0), then translation a.) may always be employed

(ELT a n)	a.) (aref a (1- n)) b.) (aref a n)
(SETA a n v)	a.) (setf (aref a (1- n)) v) b.) (setf (aref a n) v)
(ARRAYTYP a)	(inverse-translate-type (array-element-type a))
(ARRAYSIZE a)	a.) (array-total-size a) b.) (1- (array-total-size a))
(ARRAYORIG a)	?? always 1? BVM - "I can't imagine any use for ARRAYORIG other than as an ORIGIN argument to ARRAY, where it will be fine to throw it out; any other use is untranslatable. Well, maybe some index checker would use it, in which case zero would be a safe translation."
(COPYARRAY a)	(copy-seq a)
(ARRAYP a)	(vectorp a)

2.7 Arrays Interlisp-10 Arrays

Probably, no functions in this section need be supported by the translator. I list those not mentioned elsewhere here for completeness.

(ELTD a n)	?? BVM - "ELTD and SETD can only be used on arrays of type DOUBLEPOINTER. You could tediously translate them as index (+ n (/ (1- (length a)) 2) 1), but it doesn't seem worth it. ARRAYBEG is blatantly untranslatable."
(SETD a n v)	??
(ARRAYBEG a)	??

2.8 Hash Arrays

Interlisp Harryp's will most likely be represented as Common Lisp hash-tables even though Interlisp Harryp's support options more extensive then those of their counterparts.

BVM - "All the hash functions need to watch out for harray = NIL for the bogus SYSHASHARRAY feature. Probably a global note in the translator's guide is sufficient; anyone who actually wrote a program depending on the feature deserves to lose."

(HARRAY len)	(make-hash-table :size len :test #'eq) or (make-hash-table :size len) BVM - "In the case of HARRAY, you need to watch out for (list (harray len)) and (cons (harray len) overflow) and turn them into (make-hash-table :size len :rehash-size overflow). HARRAY all by itself is strictly speaking untranslatable, because it implicitly has overflow action ERROR."
(HASHARRAY minkeys)	"
(HASHARRAY minkeys overflow)	(make-hash-table :size minkeys :rehash-size overflow) BVM - "I believe the overflow arg to HASHARRAY is a superset of

the allowable values to :rehash-size, though the commonly-used numeric values are compatible (hasharray also supports the values ERROR and arbitrary function)."

(HASHARRAY minkeys overflow nil nil nil rehash-threshold)

(make-hash-table :size minkeys :rehash-size overflow :rehash-threshold rehash-threshold)

(HASHARRAY minkeys overflow hashbitsfns equivfn nil rehash-threshold)

(make-hash-table :size minkeys :test (get-know-test-fn hashbitsfns equivfn) :rehash-size overflow :rehash-threshold rehash-threshold)

(HARRAYSIZE harray)

??

(CLRHASH harray)

(clrhash harray)

(PUTHASH key val harray)

```
((lambda (x y z)
        (if y (setf (gethash x z) y)
            (remhash x z)))
  key val harray)
```

BVM - "This is another good place for a simplifier, since val=nil is usually a constant. (Unfortunately, you can rarely get rid of the remhash arm--only if the value being stored is a non-nil constant.)"

(GETHASH key harray)

(values (gethash key harray))
BVM - "I hope the simplifier knows about eliminating (values &) in non-mv context."

(REHASH oldharray newharray)

??

(MAPHASH harray maphfn)

```
((lambda (x y)
        (maphash #'(lambda (key val) (funcall y val key)) x))
  harray maphfn)
```

BVM - "Yet another place where a simplifier with sufficient smarts about lambdas would make the translation more pleasant in the common case where the maphfn is a lambda expression. Alternatively, arrange for the translator to manually permute the arg list."

(DMPHASH harray1 ... harrayn)

```
(progn (print `(setq harray1 ,harray1))
        ...
        (print `(setq harrayn ,harrayn)))
```

(HARRAYPROP harray prop)

??
BVM - ".. the only instance of which we can translate is (HARRAYPROP a 'NUMKEYS) => (hash-table-count a)."

(HARRAYPROP harray prop nv)

??

2.9 Numeric and Arithmetic Functions

Since Common Lisp arithmetic functions are fully generic -- the type specific Interlisp arithmetic functions pose a problem. They can either be a.) Correctly translated with a substantial cost in performance and complexity or b.) incorrectly translated to their generic counterparts. I will give translation for both possibilities.

BVM - "I suspect most people will want the type-specific operations to translate generically (in code I've looked at, I virtually always do), even though this will occasionally cause subtle bugs."

There may be redundancy in the following section for completeness.

Many of the following predicates could be simplified in a test context.

(SMALLP x)

((lambda (x) (and (typep x 'fixnum) x)) x)

(FIXP x)	((lambda (x) (and (integerp x) x)) x)
(FLOATP x)	((lambda (x) (and (floatp x) x)) x)
(NUMBERP x)	((lambda (x) (and (numberp x) x)) x)
MIN.SMALLP MAX.SMALLP	most-negative-fixnum most-positive-fixnum BVM - "MAX.SMALLP is often used as a synonym for 2^16-1, so this translation should be flagged."
MIN.FIXP MAX.FIXP	?? ??
MIN.INTEGER MAX.INTEGER	?? ?? BVM - "MIN.INTEGER & MAX.INTEGER are obviously untranslatable, but I think we've even de-documented them."
(OVERFLOW flg)	??
(IPLUS x1 ... xn)	a.) (+ (truncate x1) ... (truncate xn)) b.) (+ x1 .. xn)
(PLUS x1 .. xn)	(+ x1 ... xn)
(FPLUS x1 ... xn)	a.) (+ (float x1) .... (float xn)) b.) (+ x1 .. xn)
(IMINUS x)	a.) (- (truncate x)) b.) (- x)
(MINUS x)	(- x)
(FMINUS x)	a.) (- (float x)) b.) (- x)
(IDIFFERENCE x y)	a.) (- (truncate x) (truncate y)) b.) (- x y)
(DIFFERENCE x y)	(- x y)
(FDIFFERENCE x y)	a.) (- (float x) (float y)) b.) (- x y)
(ITIMES x1 ... xn)	a.) (* (truncate x1) ... (truncate xn)) b.) (* x1 .. xn)
(TIMES x1 .. xn)	(* x1 ... xn)
(FTIMES x1 ... xn)	a.) (* (float x1) .... (float xn)) b.) (* x1 .. xn)
(IQUOTIENT x y)	(truncate x y)
(QUOTIENT x y)	a.) ?? b.) (/ x y) -- although this is likely to be wrong more often than not BVM - "QUOTIENT -- I think it should only be translated as / in the case where you know that one of its args is floatp; usage tends not to be very consistent. So (if (or (floatp x) (floatp y)) (/ x y) (truncate x y)) is better, if ugly."
(FQUOTIENT x y)	a.) (/ (float x) (float y)) b.) (/ x y) -- fairly safe

(IGREATERP x y)	a.) (> (truncate x) (truncate y)) b.) (> x y)
(GREATERP x y) (FGREATERP x y)	(> x y) ""
(ILESSP x y)	a.) (< (truncate x) (truncate y)) b.) (< x y)
(LESSP x y) (FLESSP x y)	(< x y) ""
(IGEQ x y)	a.) (>= (truncate x) (truncate y)) b.) (>= x y)
(GEQP x y) (FGEQP x y)	(>= x y) ""
(ILEQ x y)	a.) (<= (truncate x) (truncate y)) b.) (<= x y)
(LEQP x y) (FLEQP x y)	(<= x y) ""
(IEQP x y)	a.) (= (truncate x) (truncate y)) b.) (= x y)
(EQP x y)	(= x y) Strictly incorrect, but probably good enough
(FEQP x y)	""
(IREMAINDER x y)	a.) (rem (truncate x) (truncate y)) b.) (rem x y)
(REMAINDER x y)	(rem x y)
(FREMAINDER x y)	a.) (rem (float x) (float y)) b.) (rem x y)
(IMIN x1 ... xn)	a.) (min (truncate x1) ... (truncate xn)) close, but not correct since (IMIN 1.2 1.1) returns 1.2 b.) (min x1 ... xn) BVM - "For IMIN, it happens to be a bug that (IMIN 1.2 1.1) returns 1.2, so I wouldn't sweat it. Ditto IMAX and IABS."
(MIN x1 ... xn)	(min x1 ... xn)
(FMIN x1 ... xn)	a.) (min (float x1) ... (float xn)) b.) (min x1 ... xn)
(IMAX x1 ... xn)	a.) (max (truncate x1) ... (truncate xn)) close, but not correct since (IMAX 1.1 1.2) returns 1.1 b.) (max x1 ... xn)
(MAX x1 ... xn)	(max x1 ... xn)
(FMAX x1 ... xn)	a.) (max (float x1) ... (float xn)) b.) (min x1 ... xn)

(IABS x)	a.) (abs (truncate x)) Not quite right, since (IABS -0.1) returns -0.1 b.) (abs x)
(ABS x)	(abs x)
(FABS x)	a.) (abs (float x)) b.) (abs x)
(ADD1 x)	(1+ x)
(SUB1 x)	(1- x)
(ZEROP x)	(zerop x)
(MINUP x)	(minusp x)
(FIX x)	(truncate x)
(GCD x y)	(gcd x y)

2.9.2 Logical Arithmetic Functions

(LOGAND x1 .. xn)	(logand x1 .. xn)
(LOGOR x1 .. xn)	(logior x1 .. xn)
(LOGXOR x1 .. xn)	(logxor x1 .. xn)
(LSH x n)	(ash x n)
(RSH x n)	(ash x (- n))
(LLSH x n)	?? usually (ash x n) will suffice
(LRSH x n)	?? usually (ash x (- n)) will suffice
(INTEGERLENGTH n)	(if (< n 0) (1+ (integer-length n) (integer-length n))
(POWEROFTWOP n)	?? roughly (zerop (logand n (1- n)))
(EVENP x)	(evenp x)
(EVENP x y)	(zerop (mod x y))
(ODDP x)	(oddp x)
(ODDP x y)	(not (zerop (mod x y)))
(LOGNOT n)	(lognot n)
(BITTEST n mask)	(logtest n mask)
(BITCLEAR n mask)	(logandc2 n mask)
(BITSET n mask)	(logior n mask)

(MASK.1'S position size)	((lambda (x y) (ldb (byte y x) -1)) position size)
(MASK.0'S position size)	((lambda (x y) (dpb 0 (byte y x) -1)) position size)
(LOADBYTE n position size)	((lambda (x y z) (ldb (byte z y) x)) n position size)
(DEPOSITBYTE n position size byte)	((lambda (w x y z) (dpb z (byte y x) w)) n position size byte)
(ROT x n fieldsize)	??
(BYTE size position)	(byte size position)
(BYTESIZE bytespec)	(byte-size bytespec)
(BYTEPOSITION bytespec)	(byte-position bytespec)
(LDB bytespec val)	(ldb bytespec val)
(DPB n bytespec val)	(dpb n bytespec val)

2.9.3 Floating Point Arithmetic

MIN.FLOAT	most-negative-single-float
MAX.FLOAT	most-positive-single-float
(FLOAT x)	(float x)

2.9.5 Special Functions

(EXPT m n)	(expt m n)
(SQRT n)	(sqrt n)
(LOG x)	(log x)
(ANTILOG x)	(exp x)
(SIN x)	(sin (degrees-to-radians x)) where (defun degrees-to-radians (degrees) (* (/ pi 180) degrees))
(SIN x t)	(sin x)
(COS x)	(cos (degrees-to-radians x))
(COS x t)	(cos x)
(TAN x)	(tan (degrees-to-radians x))
(TAN x t)	(tan x)
(ARCSIN x)	(radians-to-degrees (asin (degrees-to-radians x))) where (defun radians-to-degrees (radians) (* (/ 180 pi) radians))
(ARCSIN x t)	(asin x)

(ARCCOS x)	(radians-to-degrees (acos (degrees-to-radians x)))
(ARCCOS x t)	(acos x)
(ARCTAN x)	(radians-to-degrees (atan (degrees-to-radians x))) NB: The IRM claims the range of ARCTAN is [0, pi] -- while in the most current loadup the range is [-pi/2, pi/2]. The later situation agrees with Common Lisp.
(ARCTAN x t)	(atan x)
(ARCTAN2 x y)	(radians-to-degrees (atan (degrees-to-radians x) (degrees-to-radians y)))
(ARCTAN2 x y t)	(atan x y)
(RAND lower upper)	((lambda (x y) (+ x (random (1+ (- y x))))) lower upper) NB. The 1+ to generate an inclusive upper bound is not correct if either x or y is of type float
(RAND)	(random (1+ most-positive-fixnum))
(RANDSET X)	(defun randset (x) (case x ((t) (setq *random-state* (make-random-state))) ((nil) *random-state*) (otherwise (setq *random-state* x))**)**)

Daybreak Software Installation and Operation

This document is preliminary. It probably won't really be the way things work when the product is released.

The power switch and reset button are located on the front of the daybreak. There is no "ALT-B" button to specify alternate booting choices. Instead, the blank function keys along the top of the keyboard are used to specify the boot device.

When the daybreak is first turned on, the screen will be filled with a gray pattern with a solid white cursor in the upper left corner. Press one of the function keys according to the following table:

F1: Disk boot (Lisp)
F2: Floppy boot (doesn't work)
F3: Ethernet boot
F4: Alternate ether boot
F5: Diagnostic Disk boot
F6: Diagnostic Floppy boot
F7: Diagnostic Ether boot
F8: Reserved

Getting into Othello:

To get into othello, boot the machine then do an Alternate Etherboot-6 by pressing F4 followed by the number 6 (not F6 and not the keypad's 6). Sometime later, the cursor will change to 0900 and Othello will come up.

Installing Lisp:

The physical volume is partitioned into at least 3 volumes for Lisp.

uCode: This is where the Lisp microcode lives (about 150 pages)
Lisp: Where the sysout lives.
LispFiles: For those who want to use the local file system.

Lisp booting on the daybreak is similar to Mesa, so the microcode is stored as "Pilot Microcode" and there is a dummy "germ" file which is there just so the microcode can load something for the germ.

To bring up a new lisp on a fresh volume, do the following:

> Initial Microcode Fetch
Drive Name: RD0
File Name: <LispCore>Dove>DiskInitialDove.db

> Germ Fetch
Logical Volume Name: uCode
File Name: <LispCore>Dove>Dummy.Germ

> Lisp Microcode Fetch
Logical Volume Name: uCode
File Name: <LispCore>Dove>LispDove.db

Finally, fetching the Lisp sysout is more or less normal:

> Lisp Sysout Fetch
Logical Volume Name: Lisp

File Name: <LispCore>Next>Full.sysout
Shall I expand this lisp to fill the volume? Yes
Shall I make this the default? Yes

To boot Lisp:

Press the reset button and then press F1

Lisp caveats:

If you need the local file system, do:

(DEFINEQ (MACHINETYPE ()(QUOTE DOVE)))
then DIR {DSK} or (CREATEDSKDIRECTORY 'LISPFILES)

MACHINETYPE will return DOVE later, but the file system's eventfn crashes on daybreaks and I'm afraid to touch the file system code to disable it any other way.

LOGOUT doesn't work right. If you want to save work and resume it later, do a SAVEVM. However, if you want to resume a frozen sysout, it must be expanded and the file system must be disabled (redefine MACHINETYPE to return DAYBREAK).

If the ethernet seems to go away, use (RESTART.ETHER) to start it up again.

Lots of programs like LAFITE create their status windows off the screen.

I'm not sure what will happen if you SYSOUT from a daybreak and load into a DLion/dorado.

The Implementation of Device-Independent Graphics Through Imagestreams

filed as {Eris}<LispCore>Internal>Doc>DIGguide.TEdit
written by Herb Jellinek on 2 November 1984
last revision on 26 February 1985

The Interlisp-D system does all image creation through a set of functions and data structures for **d**evice-**i**ndependent **g**raphics, known popularly as DIG. DIG is achieved throught the use of a special flavor of stream, known as an imagestream.

An imagestream, by convention, is any stream that has its IMAGEOPS field (described in detail below) set to a vector of meaningful graphical operations. Using imagestreams, we can write programs that draw and print on an output stream without regard to the underlying device, be it window, disk, Dover, 8044 or Diablo printers. For example, the following have imagestreams backing them: windows, Press streams, Interpress streams, and Iris streams.

Imagestream structure

As indicated above, imagestreams use a field that no other stream does: IMAGEOPS. IMAGEOPS is an instance of the IMAGEOPS datatype, and contains a vector of the stream's graphical methods. The methods contained in the IMAGEOPS can make arbitrary use of the stream's IMAGEDATA field, which is provided for their use, and may contain any data needed.

IMAGEOPS [Datatype]

The IMAGEOPS datatype has the following fields:

IMAGETYPE — The name of image type. Monochrome display streams have an IMAGETYPE of DISPLAY; color display streams are identified as (COLOR DISPLAY). The IMAGETYPE is informational, and can be set to anything the implementor chooses.

IMFONTCREATE — The device name to pass to FONTCREATE when fonts are created for the stream.

The following fields are all stream methods, and are presented with their arguments, in the manner of a function definition. With the exception of IMCLOSEFN, each method that follows has a corresponding function that consists of the method's name with the "IM" prefix removed. All coordinates that refer to points in a display device's space are measured in the device's units. (The IMSCALE method provides access to a device's scale.)

(IMCLOSEFN *STREAM*)

What to do before stream is CLOSEFed, e.g. flush buffers, write header or trailer information, etc.

(IMDRAWLINE *STREAM X1 Y1 X2 Y2 WIDTH OPERATION COLOR*)

Draws a line of width *WIDTH* from (*X1*, *Y1*) to (*X2*, *Y2*). (Dashing is currently handled at a higher level, and thus is not an argument).

(IMDRAWCURVE *STREAM KNOTS CLOSED BRUSH DASHING*)

Draws a curve through *KNOTS*.

(IMDRAWCIRCLE *STREAM CENTERX CENTERY RADIUS BRUSH DASHING*)

Draws a circle of radius *RADIUS* around (*CENTERX*, *CENTERY*).

(IMDRAWELLIPSE *STREAM CENTERX CENTERY SEMIMINORRADIUS SEMIMAJORRADIUS ORIENTATION BRUSH DASHING*)

Draws an ellipse around (*CENTERX*, *CENTERY*).

(`IMFILLCIRCLE` *STREAM CENTERX CENTERY RADIUS TEXTURE*)

Draws a circle filled with texture *TEXTURE* around (*CENTERX*, *CENTERY*).

(`IMBLTSHADE` *TEXTURE STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT OPERATION CLIPPINGREGION*)

The texture-source case of `BITBLT`. *DESTINATIONLEFT*, *DESTINATIONBOTTOM*, *WIDTH*, *HEIGHT,* and *CLIPPINGREGION* are measured in *STREAM*'s units. This method is invoked by the functions `BITBLT` and `BLTSHADE`.

(`IMBITBLT` *SOURCEBITMAP SOURCELEFT SOURCEBOTTOM STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION CLIPPEDSOURCELEFT CLIPPEDSOURCEBOTTOM*)

The bitmap-source cases of `BITBLT`. *SOURCELEFT*, *SOURCEBOTTOM*, *CLIPPEDSOURCELEFT*, *CLIPPEDSOURCEBOTTOM*, *WIDTH*, and *HEIGHT* are measured in pixels; *DESTINATIONLEFT*, *DESTINATIONBOTTOM*, and *CLIPPINGREGION* are in the units of the destination stream.

(`IMSCALEDBITBLT` *SOURCEBITMAP SOURCELEFT SOURCEBOTTOM STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION CLIPPEDSOURCELEFT CLIPPEDSOURCEBOTTOM SCALE*)

A scaled version of `IMBITBLT`. Each pixel in *SOURCEBITMAP* is replicated *SCALE* times in the X and Y directions; currently, *SCALE* must be an integer.

(`IMMOVETO` *STREAM X Y*)

Move to (*X*,*Y*). This method is invoked by the functions `MOVETO` and `RELMOVETO`. If it is not supplied, a default method composed of calls to the `IMXPOSITION` and `IMYPOSITION` methods is used.

(`IMTERPRI` *STREAM*)

(As yet unused.) Issue a newline. When implemented, this method will be invoked by the function `TERPRI`. It defaults to `(\OUTCHAR` *STREAM* `(CHARCODE EOL))`.

(`IMSTRINGWIDTH` *STREAM STR RDTBL*)

Returns the width of string *STR* in *STREAM*'s units, using *STREAM*'s current font. If this method is not supplied, it defaults to calling `\STRINGWIDTH.GENERIC`.

(`IMCHARWIDTH` *STREAM CHARCODE*)

Returns the width of character *CHARCODE* in *STREAM*'s units. If this method is not supplied, it defaults to calling `\STRINGWIDTH.GENERIC`.

The following methods all have corresponding `DSPxx` functions (e.g. `IMYPOSITION` corresponds to `DSPYPOSITION`) that invoke them. They also have the property that they return their previous value; when called with `NIL` they return the old value without changing it.

(`IMXPOSITION` *STREAM XPOSITION*)

Sets new x-position on *STREAM*.

(`IMYPOSITION` *STREAM YPOSITION*)

Sets new y-position on *STREAM*.

(`IMFONT` *STREAM FONT*)

Sets *STREAM*'s font to be *FONT.*

`(IMLEFTMARGIN` *STREAM LEFTMARGIN*`)`

Sets *STREAM*'s left margin to be *LEFTMARGIN*. The left margin is defined as the x position set after newline.

`(IMRIGHTMARGIN` *STREAM RIGHTMARGIN*`)`

Sets *STREAM*'s right margin to be *RIGHTMARGIN*. The right margin is defined as the maximum x position at which characters will be printed; printing beyond it causes a newline.

`(IMLINEFEED` *STREAM DELTA*`)`

Sets *STREAM*'s linefeed distance (distance to move vertically after a newline) to be *DELTA*.

`(IMNEWPAGE` *STREAM*`)`

Causes a new page to be started; the position is set to (*DSPLEFTMARGIN*, *DSPTOPMARGIN*). If not supplied, defaults to `(\OUTCHAR` *STREAM* `(CHARCODE ^L))`.

`(IMSCALE` *STREAM SCALE*`)`

Returns the number of device points per screen point (a screen point being ~1/72 inch). In a later release of Interlisp-D the conversion factor will be specifiable. (I.e. right now *SCALE* is ignored.)

`(IMTOPMARGIN` *STREAM YPOSITION*`)`

Sets *STREAM*'s top margin (the y-position of the tops of characters that is set after newpage) to be *YPOSITION*.

`(IMBOTTOMMARGIN` *STREAM YPOSITION*`)`

Sets *STREAM*'s bottom margin (the y-position beyond which any printing causes a newpage) to be *YPOSITION*.

`(IMSPACEFACTOR` *STREAM FACTOR*`)`

Sets the amount by which to multiply the natural width of all following space characters on *STREAM*: used for justification of text. The default value is 1. For example, if the natural width of a space in *STREAM*'s current font is 12 units, and the spacefactor is set to 2, spaces will appear 24 units wide. The values returned by `STRINGWIDTH` and `CHARWIDTH` will also be affected.

`(IMOPERATION` *STREAM OPERATION*`)`

Sets the default `BITBLT` *OPERATION* argument. See the `DSPOPERATION` and `BITBLT` documentation for more information.

`(IMBACKCOLOR` *STREAM COLOR*`)`

Sets the background color of *STREAM*.

`(IMCOLOR` *STREAM COLOR*`)`

Sets the default color of *STREAM*.

In addition to the `IMAGEOPS`-borne methods described above, there are two other important methods, which are contained in the stream itself.

`STRMBOUTFN` [Stream Method]

Function called by BOUT. You can install a STRMBOUTFN in a stream *STREAM* using the form (**replace** (STREAM STRMBOUTFN) **of** STREAM **with** (FUNCTION MYBOUTFN)).

OUTCHARFN [Stream Method]

This is the function that is called to output a single byte. This is like STRMBOUTFN, except for being one level higher: it is intended for text output. Hence, this function should convert (CHARCODE EOL) into the stream's actual end of line sequence, and should adjust the stream's CHARPOSITION appropriately before invoking the stream's STRMBOUTFN (by calling BOUT) to actually put the character. Defaults to \FILEOUTCHARFN, which is definitely NOT what you want. OUTCHARFNs are installed using a form like (**replace** (STREAM OUTCHARFN) **of** STREAM **with** (FUNCTION MYOUTCHARFN)).

IMAGEDATA [Record field]

Used to hold data pertaining to this type of imagestream; the content is completely up to the implementor. For Interpress devices, this is an instance of the datatype INTERPRESSDATA; for Press, PRESSDATA; for the display, \DISPLAYDATA.

Creating imagestreams

(OPENIMAGESTREAM *FILE IMAGETYPE OPTIONS*) [function]

Opens and returns an output stream of type *IMAGETYPE* (PRESS, INTERPRESS, DISPLAY or other types) on a destination specified by *FILE*. *FILE* can name a file either on a normal storage device or on a printer device. In the latter case, the file is sent to the printer when the stream is closed. Because of the way that defaulted arguments are interpreted, OPENIMAGESTREAM provides a convenient and standard interface for interpreting user output-destination specifications.

If *IMAGETYPE* is NIL, the image type is inferred from the extension field of *FILE* and the EXTENSIONS properties in the list PRINTFILETYPES. Thus, a PRESS extension denotes a Press-format stream, while IP, IPR, and INTERPRESS indicate Interpress format. If *FILE* has no extension but is a file on the printer device {LPT}, then *IMAGETYPE* will be the type that the indicated printer can print. If *FILE* has no extension but is not on the printer device, then *IMAGETYPE* will default to the type accepted by the first printer on DEFAULTPRINTINGHOST.

FILE = NIL is equivalent to *FILE* = {LPT}. Names for printer files are of the form {LPT}PRINTERNAME.TYPE, where PRINTERNAME, TYPE, or both may be omitted. PRINTERNAME is the name of the particular printer to which the file will be transmitted on closing; it defaults to the first printer on DEFAULTPRINTINGHOST that can print *IMAGETYPE* files. As just described, the TYPE extension supplies the *IMAGETYPE* when it is defaulted. OPENIMAGESTREAM will generate an error if the specified printer does not accept the kind of file specified by *IMAGETYPE*.

Examples assuming IP: is an Interpress printer, P is a Press printer, and DEFAULTPRINTINGHOST is (IP: P):

(OPENIMAGESTREAM) returns an Interpress image stream on printer IP:

(OPENIMAGESTREAM NIL 'PRESS) returns a Press stream on P

(OPENIMAGESTREAM '{LPT}.INTERPRESS) returns an Interpress stream on IP:

(OPENIMAGESTREAM '{CORE}FOO.PRESS) returns a Press stream on the file {CORE}FOO.PRESS

For completeness and consistency, if *IMAGETYPE* is inferred to be DISPLAY, then the user is prompted for a window to open. The file name in this case will be used as the title of the window.

OPTIONS is a list in property list format that may be used to specify certain attributes of the image stream; not all attributes are meaningful or interpreted by all types of streams. Among the properties are:

REGION value is the region on the page (in stream scale units, 0,0 being the lower-left corner of the page) that text will fill up. It establishes the initial values for `DSPLEFTMARGIN`, `DSPRIGHTMARGIN`, `DSPBOTTOMMARGIN` and `DSPTOPMARGIN`.

FONTS value is a list of fonts that are expected to be used in the stream. Some streams (e.g. Interpress) are more efficient if the expected fonts are called out in advance, but this is not necessary. The first font in this list will be the initial font of the stream, otherwise the `DEFAULTFONT` for that image type will be used.

HEADING the heading to be placed automatically on each page, `NIL` means no heading.

`IMAGESTREAMTYPES` [a-list]

Describes how to create a stream for a given image type. Contains `OPENSTREAM`, `FONTCREATE`, `FONTSAVAILABLE` methods. The main a-list is indexed by the image-stream type name (e.g., `DISPLAY`, `PRESS`, or `INTERPRESS`) to get another a-list that associates device-dependent functions with generic operation names.

Format of a single a-list entry:

```
(imagetype
  (OPENSTREAM function-to-open-the-stream)
  (FONTCREATE function-to-create-a-fontdescriptor)
  (FONTSAVAILABLE function-to-return-available-fonts))
```

For example, for Interpress, the a-list entry is:

```
(INTERPRESS
  (OPENSTREAM OPENIPSTREAM)
  (FONTCREATE \CREATEINTERPRESSFONT)
  (FONTSAVAILABLE \SEARCHINTERPRESSFONTS))
```

The `OPENSTREAM` function is called with arguments:

(`openstreamfn` *FILE OPTIONS*)

> *FILE* is the file name as it was passed to `OPENIMAGESTREAM`, and *OPTIONS* is the *OPTIONS* property list passed to `OPENIMAGESTREAM`. The result must be a stream of the appropriate imagetype.

The `FONTCREATE` function is called with arguments:

(`fontcreatefn` *FAMILY SIZE FACE ROTATION DEVICE*)

> *FAMILY* is the family name for the font, e.g. `MODERN`. *SIZE* is the body size of the font, in printer's points. *FACE* is a 3-element list describing the weight, slope, and expansion of the face desired, e.g. `(MEDIUM ITALIC EXPANDED)`. *ROTATION* is how much the font is to be rotated from the normal orientation, in minutes of arc. For example, to print a landscape page, fonts would have rotation 5400 (=90 degrees). The function's result must be a `FONTDESCRIPTOR` with the fields filled in appropriately.

The `FONTSAVAILABLE` function is called with arguments:

(`fontsavailablefn` *FAMILY SIZE FACE ROTATION DEVICE*)

> This function returns a list of all fonts agreeing with the *FAMILY*, *SIZE*, *FACE*, and *ROTATION* arguments; any of them may be wildcarded (i.e. equal to `'*`, which means "any"). Each

element of the list should be a 5-tuple of the form (`FAMILY SIZE FACE ROTATION DEVICE`).

Where the function looks is an implementation decision: the `fontsavailablefn` for the display device looks at `DISPLAYFONTDIRECTORIES`, the Interpress code looks on `INTERPRESSFONTDIRECTORIES`, and implementors of new devices should feel free to introduce new search path variables.

Imagestream predicates

(`IMAGESTREAMP` *X IMAGETYPE*) [function]

Returns *X* (possibly coerced to a stream) if it is an output image stream of type *IMAGETYPE* (or of any type if *IMAGETYPE* `= NIL`), otherwise `NIL`.

(`IMAGESTREAMTYPE` *STREAM*) [function]

Returns the image type of *STREAM*.

(`IMAGESTREAMTYPEP` *STREAM TYPE*) [function]

Returns T if *STREAM* is an imagestream of type *TYPE*.

Creating your own flavor of imagestream

In accomplishing a task as complex as building a new flavor of imagestream, no document can contain all of the answers, tricks, or shortcuts. There is no substitute for studying a working implementation in doing your own. Therefore, we recommend you look at the `FX80STREAM` package as an example of how to create a new imaging device. `FX80STREAM` is a `DIG` interface for the Epson FX-80 printer - a device simple enough to drive that its details will not obscure the fundamentals of how its imagestream works.

The dld-link SUBR opcode

This opcode links in a relocaltable object file or a library file. The return value is an error code.

`(SUBR DLD-LINK` *path*`)`

Path is the path of the file to be loaded expressed as a string.

Return-Code is a FIXP cell into which DLD-LINK places a return value. See below for possible values.

Opcode is #o250

The dld-unlink-by-file SUBR opcode

This opcode unlinks in a relocaltable object file or a library file. The return value is an error code.

`(SUBR DLD-UNLINK-BY-FILE` *string hard-flag*`)`

String is the name of the file to be unlinked.
Hard-flag is zero if the file shouldn't be unlinked if it is referenced by others. A nonzero value means that the file should be unlinked regardless of other references.

Return-Code is a FIXP cell. See below for possible values.

Opcode is #o251

The dld-unlink-by-symbol SUBR opcode

This opcode unlinks in a relocaltable object file or a library file. The return value is an error code.

(SUBR DLD-UNLINK-BY-SYMBOL *string*)

String is the name of a symbol containde in the file to be unlinked.
Hard-flag is zero if the file shouldn't be unlinked if it is referenced by others. A nonzero value means that the file should be unlinked regardless of other references.

Return-Code is a FIXP cell. See below for possible values.

Opcode is #o252

The dld-get-symbol SUBR opcode

This opcode returns a pointer to a symbol in C-space.

`(SUBR DLD-GET-SYMBOL` *string*`)`

String is the name of a symbol.

Opcode is #o253

The dld-get-func SUBR opcode

This opcode returns a pointer to a function in C-space.

`(SUBR DLD-GET-FUNC` *string*`)`

String is the name of a function.

Opcode is #o254

The dld-function-executable-p SUBR opcode

This opcode returns 0 if the function contains undefined references.

`(SUBR DLD-FUNCTION-EXECUTABLE-P` *string*`)`

String is the name of a function.

Opcode is #o255

The dld-list-undefined-sym SUBR opcode

This opcode returns returns a list of the undefined symbols in the system.

```
(SUBR DLD-LIST-UNDEFINED-SYM)
```

Opcode is #o256

The dld-create-reference SUBR opcode

This opcode forces the system to load the library containing this symbol.

(SUBR DLD-CREATE-REFERENCE *string*)

```
0       Successful call and return.
-1      DLD-ENOFILE    cannot open file.
-2      DLD-EBADMAGIC  bad magic number.
-3      DLD-EBADHEADER failiure reading header.
-4      DLD-ENOTEXT    premature end of file in text section.
-5      DLD-ENOSYSMBOLS        premature end of file in symbol section.
-6      DLD-ENOSTRINGS bad string table.
-7      DLD-ENOTEXTRELOC       premature end of file in text relocation.
-8      DLD-ENODATA    premature end of file in data section.
-9      DLD-ENODATARELOC       premature end of file in data relocation.
-10     DLD-EMULTDEFS  multiple definitions of symbol.
-11     DLD-EBADLIBRARY        malformed library archive.
-12     DLD-EBADCOMMON common block not supported.
-13     DLD-EBADOBJECT malformed input file.
-14     DLD-EBADRELOC bad relocation info.
-15     DLD-ENOMEMORY  vmem exhausted.
-16     DLD-EUNDEFSYM  undefined symbol.
```

Fasl format change log.

Version 3 (10-Nov-86)
  Added opcode 149, FASL-INTERLISP-SYMBOL pname(V).
  Upward compatible with version 2.

FLOPPY

INTRODUCTION

You are reading the Introduction. First comes a section on the low level 1108 floppy implementation. This includes discussion of FLOPPYIOCBs, DISKADDRESSes, and FLOPPYRESULTs. The low level floppy command functions are described in this section.

Next comes a section on Pilot floppy basics. This section describes file device \PFLOPPYFDEV and the functions installed on \PFLOPPYFDEV which are called by generic FILEIO functions. We describe the peculiar marker pages (PMPAGEs)...

LOW LEVEL FLOPPY

This section describes the lowest level code of the 1108 FLOPPY implementation.

Miscellaneous Global Variables

\FLOPPY.CYLINDERS **(Variable)**

\FLOPPY.TRACKSPERCYLINDER **(Variable)**

\FLOPPY.SECTORSPERTRACK **(Variable)**

The number of cylinders on the floppy, the number of tracks per cylinder on the floppy, and the number of sectors per track in the DATA part of the floppy. On the 1108, \FLOPPY.CYLINDERS=77, \FLOPPY.TRACKSPERCYLINDER=2, and \FLOPPY.SECTORSPERTRACK=15. Some of these values are different for the 1186.

\FLOPPYIOCB and 1108 Input Output Control Blocks

\FLOPPYIOCB **(Variable)**

Points to the 1108 floppy input output control block. Lisp is run by the CP Central Processor and must communicate via the input output control block to the IOP Input Output Processor the various floppy commands that need to be executed.

It will appear that \FLOPPYIOCB is bound to CURSORBITMAP.EM. This is just a peculiarity of how the Lisp virtual memory works.

An input output control block looks like

FLOPPYIOCB **(Datatype)**

```
(DATATYPE FLOPPYIOCB
  ((\BUFFERLOLOC WORD)
   (\BUFFERHILOC WORD)
   (NIL WORD)
   (SECTORLENGTHDIV2 WORD)
   (TROYORIBM BITS 12)
   (DENSITY BITS 4)
   (DISKADDRESS FIXP)
   (SECTORCOUNT WORD)
   (FLOPPYRESULT WORD)
   (SAMEPAGE FLAG)
   (COMMAND BITS 15)
   (SUBCOMMAND WORD)
   (SECTORLENGTHDIV4 BITS 8)
   (ENCODEDSECTORLENGTH BITS 8)
   (SECTORSPERTRACK BITS 8)
   (GAP3 BITS 8)
   (NIL 3 WORD)))
```

The Lisp code sets up and the IOP reads all fields except for FLOPPYRESULT. The IOP sets and the Lisp code reads the FLOPPYRESULT.

\BUFFERHILOC and \BUFFERLOLOC are the high and low part of the pointer to the page that is being read from or being written to if any. We are not able to put (BUFFER POINTER) instead of (\BUFFERLOLOC WORD) and (\BUFFERHILOC WORD) because \BUFFERHILOC and \BUFFERLOLOC occur in the wrong order. The reason for this is that Mesa pointers are swapped and Lisp pointers are not.

SECTORLENGTHDIV2 is the length in bytes of the sectors being read or written divided by 2.

TROYORIBM is always IBM.

DENSITY is usually DOUBLE. DENSITY is SINGLE for track CYLINDER=0, HEAD=0 on Pilot floppies.

DISKADDRESS is a FIXP encoding a CYLINDER+HEAD+SECTOR combination. Each DISKADDRESS points to a page (aka sector) on the floppy. The particular format of DISKADDRESSes is described elsewhere in this document.

SECTORCOUNT is looked at by the IOP only when the COMMAND is to format the floppy. SECTORCOUNT indicates how many tracks to format.

FLOPPYRESULT is the status result word which is set by the IOP. The FLOPPYRESULT is a set of flags which can be usefully inspected with the FLOPPYRESULT BLOCKRECORD described elsewhere in this document.

SAMEPAGE is always NIL.

COMMAND can be any of the following constants

C.NOP	**(Constant)**
C.READSECTOR	**(Constant)**
C.WRITESECTOR	**(Constant)**
C.FORMATTRACK	**(Constant)**
C.RECALIBRATE	**(Constant)**

C.INITIALIZE **(Constant)**

SUBCOMMAND is always SC.NOP.

GAP3, SECTORLENGTHDIV4, ENCODEDSECTORLENGTH, and SECTORSPERTRACK are obscure numbers that need to be set up for the IOP when COMMAND is to format.

1108 Floppy DISKADDRESSes

Locations of pages on a floppy are referrred to by the hardware by cylinder+head+sector number combinations called DISKADDRESSes. DISKADDRESSes are represented by FIXPs and are accessed and created with the help of the record declaration

DISKADDRESS **(Accessfns)**

```
(ACCESSFNS DISKADDRESS
  ((CYLINDER (LRSH DATUM 16))
   (HEAD (LRSH (LOGAND DATUM 65535) 8))
   (SECTOR (LOGAND DATUM 255)))
   (CREATE (IPLUS (LLSH CYLINDER 16)
                  (LLSH HEAD 8)
                  SECTOR)))
```

CYLINDER can be between 0 and \FLOPPY.CYLINDERS.

HEAD can be 0 or 1 to indicate which side of the floppy should be read.

The part of the floppy pointed to by a CYLINDER+HEAD combination is known as a track. Each track contains a certain amount of redundant operation for self checking puposes conducted by the IOP assembly language code plus actual data stored in pages called sectors.

SECTOR can be between 1 and 15 for tracks that are formatted IBM double density 512 bytes per sector. The upper limit for SECTOR is determined according to the formatting of the track as given by the following table

Format	Number of Sectors
IBMS128	26
IBMS256	15
IBMS512	8
IBMS1024	4
IBMD128	36
IBMD256	26
IBMD512	15
IBMD1024	8

The table above can be found coded into the function \FLOPPY.SECTORSPERTRACK.

Pilotconverts between Pilot page numbers and DISKADDRESSes by using the functions \PFLOPPY.PAGENOTODISKADDRESS and \PFLOPY.DISKADDRESSTOPAGENO.

\FLOPPYRESULT and 1108 Result Words

\FLOPPYRESULT **(Variable)**

Points to the 1108 floppy result word offsetted into the \FLOPPYIOCB input output control block. Lisp is run by the CP Central Processor and must communicate via the input output control block to the IOP Input Output Processor the various floppy commands that need to be executed. After a command is executed, the Lsip code looks at the status word to see if any error conditions were signalled indicating a failed operation.

It will appear that \FLOPPYRESULT is bound to MOUSEX.EM.This is just a peculiarity of how the Lisp virtual memory works.

An 1108 result word looks like

FLOPPYRESULT **(Blockrecord)**

```
(BLOCKRECORD FLOPPYRESULT
  ((DOOROPENED FLAG)
   (MPERROR FLAG)
   (TWOSIDED FLAG)
   (DISKID FLAG)
   (ERROR FLAG)
   (NIL FLAG)
   (RECALIBRATEERROR FLAG)
   (DATALOST FLAG)
   (NOTREADY FLAG)
   (WRITEPROTECT FLAG)
   (DELETEDDATA FLAG)
   (RECORDNOTFOUND FLAG)
   (CRCERROR FLAG)
   (TRACK0 FLAG)
   (NIL FLAG)
   (BUSY FLAG)))
```

It should be pointed out that some of the error bits in the \FLOPPYRESULT are a bit noisy. Certain errors turn out to be transients that can be overcome simply by reissuing the last command. The bits MPERROR and CRCERROR are notably noisy. The Lisp code takes the first occurence of certain errors as an indication that perhaps the floppy drive has lost its way and needs to be recalibrated. In that case, a recalibrate command is issued and then the command that failed is tried again. After sufficient retrying of a command that fails (the amount of retrying depending on the kind of command and the particular settings of the error bits), the bad news is announced to the user. Most of this retrying and so on has been determined empirically and is coded into the function \FLOPPY.RUN.

DOOROPENED is T if the floppy drive door has opened since the last floppy command. This is an error condition which has to be cleared. (When DOOREOPENED gets set, ERROR also gets set.) To clear this bit, an INITIALIZE command must be issued.

MPERROR is T if the IOP floppy handler tried to crash the machine. We suspect this condition gets set from time to time because of bugs in the IOP assembly language code written by OSD. The MPERROR feature is a patch on top of the OSD assembly language code installed by Mitch Lichtenberg. Instead of crashing the machine the flag MPERROR now gets set and Lisp deals with that. It has turned out to be the case that reissuing the last command to the IOP without change invariably works. When MPERROR is set to T, the remainder of the FLOPPYRESULT word is treated as an error code instead of the flags DOOROPENED, TWOSIDED,DISKID, ... BUSY. The MPCODEs stored in the remainder of the FLOPPYRESULT word can be

```
580 Domino NoValidCommand Error
581 Domino UnImplFloppyCmd Error
582 Domino InvalidEscapeCmd Error
583 Domino CommandTrack Error
584 Domino TrackToBig Error
585 Domino BadDmaChannel Error
586 Domino NoDmaEndCount1 Error
587 Domino NoDmaEndCount2 Error
597 Domino Error In NOOP Patch
598 Domino Error in Reset Patch
```

TWOSIDED is T if the floppy is two sided.

DISKID = T if the floppy drive is Schogart Associated model SA850 and NIL if the floppy drive is Schogart Associated model SA800.

ERROR = T if there was an error in trying to perofrm the last command issued to the IOP. To clear this bit, an INITIALIZE command must be issued.

RECALIBRATEERROR = T if there was an error while recalibrating I suppose. I don't think I've ever seen this flag set in practice except perhaps if you issue a recalibrate command when there is no floppy in the floppy drive. The Lisp code in that situation would tend to error out on a command preceding any recalibration.

DATALOST = T problem reading or writing. Maybe the track of the floppy which is being read from or written into isn't formatted in the way that was expected.

NOTREADY = T implies the IOP is not ready I guess. I'm not sure I've ever seen this.

WRITEPROTECT = T means the floppy is writeprotected. Any attempt to write on the floppy will fail so the Lisp code checks whether this flag is on and signals an error if need be before attempting to open an ouptut stream or format a floppy.

DELETEDDATA = I have no idea.

RECORDNOTFOUND = T problem reading or writing. Maybe the track of the floppy which is being read from or written into isn't formatted in the way that was expected.

CRCERROR = Cyclic Redundancy Check. There is redundant information on the floppy which serves as a check to the IOP assembly language code that the floppy drive head is where it is supposed to be. This error bit is a bit noisy and spurious CRCERRORs can be overcome by reissuing the last command to the IOP. If after several retries the CRCERROR has not gone away, then the CRCERROR is treated as real and the user is hit with the bad news. A hard CRCERROR is caused by the track of the floppy which is being read from or written into not being formatted in the way that was expected.

TRACK0 = T after a recalibrate command. Says that the recalibrate successfully found CYLINDER=0 I guess. Not really paid attention to by the Lisp code.

BUSY = T implies the IOP is busy I guess. I'm not sure if this would mean that the IOP is still preocessing the most recently issued command or not. The official way to wait for the IOP to finish the most recently issued command is with the loop

(while (NOT (ZEROP (fetch (IOPAGE DLFLOPPYCMD) of \IOPAGE))) do (BLOCK))

Low Level 1108 Floppy Command Functions

(\FLOPPY.RUN FLOPPYIOCB NOERROR)

This function makes the IOP really go.

FLOPPYIOCB should already be prepared by other functions like \FLOPPY.NOP, \FLOPPY.READSECTOR, \FLOPPY.WRITESECTOR, \FLOPPY.FORMATTRACKS, \FLOPPY.RECALIBRATE, or \FLOPPY.INITIALIZE described below.

First, if there is a buffer involved (the command is to read or to write), the buffer is locked down by calling the function \FLOPPY.LOCK.BUFFER.

Next, the FLOPPYIOCB passed in is \BLTed on to \FLOPPYIOCB. \FLOPPYIOCB points to the beginning of the 16 words in real memory that the IOP will look at and interpret as an input output control block to be processed.

Next, the IOP is notified that there is an input output control block in need of processing via

(replace (IOPAGE DLFLOPPYCMD) of \IOPAGE with \FLOPPYIOCBADDR)

The IOP periodically looks at the \IOPAGE for things to do and acts when it sees a nonzero DLFLOPPYCMD field in the \IOPAGE. After replacing the DLFLOPPYCMD field in the \IOPAGE, the Lisp code must wait until the IOP finishes via the loop

(while (NOT (ZEROP (fetch (IOPAGE DLFLOPPYCMD) of \IOPAGE))) do (BLOCK))

After this loop finishes, \FLOPPY.RUN looks at the \FLOPPYRESULT result word to see if any error flags have been set by the IOP. Supposing things have gone well, \FLOPPY.UNLOCK.BUFFER is called to unlock the buffer (if any) pointed to by the FLOPPYIOCB and \FLOPPY.RUN returns T indicating success.

If error bits in the \FLOPPYRESULT result word have been set, then \FLOPPY.RUN may try to recover in certain ways. This may involve reissuing a command and/or some intervening recalibration commands.

If an error persists, then \FLOPPY.RUN returns NIL if NOERROR = T and otherwise a break and an error message happen to the user.

(\FLOPPY.LOCK.BUFFER FLOPPYIOCB)

Calls \LOCKPAGES on the buffer pointed to by the FLOPPYIOCB \BUFFERLOLOC and \BUFFERHILOC fields. It is necessary that the buffer be locked down before a command can be issued to the IOP. This function also does \GETBASEing and \PUTBASEing into the BUFFER to make the BUFFER look dirty to the IOP. This is known as "touching pages". A reasonable person would think that this touching ought not to be forced on the Lisp floppy handler, but that's just the way it is. If the Lisp floppy handler doesn't touch the buffer, the IOP will sometimes cause a fatal 510 crash. \FLOPPY.LOCK.BUFFER gets called by \FLOPPY.RUN before a command to the IOP is allowed to begin exectuing.

(\FLOPPY.UNLOCK.BUFFER FLOPPYIOCB)

Calls \UNLOCKPAGES on the buffer pointed to by the FLOPPYIOCB. \FLOPPY.UNLOCK.BUFFER gets called by \FLOPPY.RUN after a command to the IOP has finished executing.

(\FLOPPY.NOP NOERROR)

Calls \FLOPPY.RUN to perform a NOP command. This command can be used to get the state of the DOOROPENED and WRITEPROTECT flags of the \FLOPPYRESULT without doing actual operations.

(\FLOPPY.READSECTOR FLOPPYIOCB DISKADDRESS PAGE NOERROR)

Calls \FLOPPY.RUN to perform a read. PAGE should be a virtual memor page VMEMPAGEP. Typically PAGE is a virtual memory page that has worked its way down from FILEIO functions to \FLOPPY.READSECTOR via the functions \PFLOPPY.READPAGES and \PFLOPPY.READPAGENO. FLOPPYIOCB will normally be \FLOPPY.IBMD512.FLOPPYIOCB. Data at the page of the floppy located at DISKADDRESS is read into PAGE.

(\FLOPPY.WRITESECTOR FLOPPYIOCB DISKADDRESS PAGE NOERROR)

Calls \FLOPPY.RUN to perform a write. PAGE should be a virtual memor page VMEMPAGEP. Typically PAGE is a virtual memory page that has worked its way down from FILEIO functions to \FLOPPY.WRITESECTOR via the functions \PFLOPPY.WRITEPAGES and \PFLOPPY.WRITEPAGENO. FLOPPYIOCB will normally be \FLOPPY.IBMD512.FLOPPYIOCB. The contents of PAGE are written into the page of the floppy located at DISKADDRESS.

(\FLOPPY.FORMATTRACKS FLOPPYIOCB DISKADDRESS KOUNT NOERROR)

Calls \FLOPPY.RUN to perform formatting. This function will format KOUNT tracks on the

(fetch (DISKADDRESS HEAD) of DISKADDRESS)

side of the floppy beginning with cylinder

(fetch (DISKADDRESS CYLINDER) of DISKADDRESS)

The quantity (fetch (DISKADDRESS SECTOR) of DISKADDRESS) is ignored for purposes of formatting.

KOUNT is spelled with a K just because COUNT is a CLISP word and CLISP gets in your way if you spell the word with a C.

The kind of formatting that takes place depends on the kind of FLOPPYIOCB. If FLOPPYIOCB is \FLOPPY.IBMD512.FLOPPYIOCB, then tracks are formatted IBM double density 512 bytes per sector. If FLOPPYIOCB is \FLOPPY.IBMD256.FLOPPYIOCB, then tracks are formatted IBM double density 256 bytes per sector.If FLOPPYIOCB is \FLOPPY.IBMS128.FLOPPYIOCB, then tracks are formatted IBM single density 128 bytes per sector.

Pilot floppies are formatted this way:

CYLINDER=0, HEAD=0 => IBMS128 format

CYLINDER=0, HEAD=1 => IBMD256 format

CYLINDER >0, HEAD=0 or 1 => IBMD512 format

(\FLOPPY.RECALIBRATE NOERROR)

This function is called to ask the floppy drive hardware to recalibrate itself. The floppy drive head positions itself over tracks on a floppy with the aid of a stepping motor which steps in from CYLINDER=0 which is specially recognizable. The stepping motor can slip some and it is occasionally necessary to ask the floppy drive hardware to recalibrate itself which means refind CYLINDER=0. CYLINDER = 0 is kind of a light pole on a very dark street for the floppy hardware. Once the hardware is back in the light of CYLINDER = 0, it knows where it is again.

\FLOPPY.RECALIBRATE is sometimes called by the Lisp implementation after certain errors are detected while performing normal commands like reading and writing. The assumption in those situations is that the stepping motor has slipped some and that recalibrating will make things well enough again that the command which failed will succeed if it is reissued after the recalibration. (Of course, after enough times of trying this strategy, the error just has to be announced to the user.)

\FLOPPY.RECALIBRATE is also called at the beginning of certain major operations. For example, \FLOPPY.RECALIBRATE is called several times by the formatting function \PFLOPPY.FORMAT. After all, if we're going to format a floppy which involves writing on to the floppy, we want to do the best job we can.

(\FLOPPY.INITIALIZE NOERROR)

\FLOPPY.INITIALIZE initializes the IOP assembly language code floppy handler which Lisp must talk with. This function has to be the first function that is called when FLOPPY is started up. Thus, \FLOPPY.EVENTFN calls \FLOPPY.INITIALIZE after coming back from a LOGOUT, SYSOUT, MAKESYS, or SAVEVM.

\FLOPPY.INITIALIZE also has to be called to clear error conditions as they arise. This might happen, for example, if the user trys to read from the floppy drive but the floppy drive door is open or there is no floppy. This would set the DOOROPENED error bit in the

\FLOPPYRESULT word. To clear these error conditions like DOOROPENED, \FLOPPY.INITIALIZE needs to be called.

PILOT FLOPPY

This section describes the connection between FLOPPY and FILEIO. FILEIO is the system source file that makes device independent part of operations like OPENSTREAM, READ, PRINT, CLOSEF, RENAMEFILE, DELFILE, and DIRECTORY possible. FILEIO ultimately winds up calling functions stored in the fields of file device records.

\PFLOPPYFDEV

\PFLOPPYFDEV (Variable)

The major FDEV record for floppy is bound to \PFLOPPYFDEV. At the time of this writing, \PFLOPPYFDEV looks like the following image

```
{FDEV}#66,3400 Inspector
-----------------------------------------------------------------------------
-----------------
DEVICENAME          FLOPPY
RESETABLE           T
RANDOMACCESSP       T
NODIRECTORIES       T
PAGEMAPPED          T
FDBINABLE           T
FDBOUTABLE          T
FDEXTENDABLE        T
BUFFERED            T
REMOTEP             NIL
SUBDIRECTORIES      NIL
CLOSEFILE           \PFLOPPY.CLOSEFILE
DELETEFILE          \PFLOPPY.DELETEFILE
DIRECTORYNAMEP      TRUE
EVENTFN             \FLOPPY.EVENTFN
GENERATEFILES
\PFLOPPY.GENERATEFILES
GETFILEINFO         \PFLOPPY.GETFILEINFO
GETFILENAME         \PFLOPPY.GETFILENAME
HOSTNAMEP           \FLOPPY.HOSTNAMEP
OPENFILE            \PFLOPPY.OPENFILE
READPAGES           \PFLOPPY.READPAGES
REOPENFILE          \PFLOPPY.OPENFILE
SETFILEINFO         \PFLOPPY.SETFILEINFO
TRUNCATEFILE        \PFLOPPY.TRUNCATEFILE
WRITEPAGES          \PFLOPPY.WRITEPAGES
BIN                 \BUFFERED.BIN
BOUT                \BUFFERED.BOUT
PEEKBIN             \BUFFERED.PEEKBIN
READP               \PAGEDREADP
BACKFILEPTR         \PAGEDBACKFILEPTR
DEVICEINFO          {PINFO}#65,4764
FORCEOUTPUT         \PAGED.FORCEOUTPUT
LASTC               NIL
SETFILEPTR          \PAGEDSETFILEPTR
GETFILEPTR          \PAGEDGETFILEPTR
GETEOFPTR           \PAGEDGETEOFPTR
```

EOFP	\PAGEDEOFP
BLOCKIN	\BUFFERED.BINS
BLOCKOUT	\BUFFERED.BOUTS
RENAMEFILE	\PFLOPPY.RENAMEFILE
RELEASEBUFFER	NIL
GETNEXTBUFFER	\PAGED.GETNEXTBUFFER
SETEOFPTR	\PAGED.SETEOFPTR
FREEPAGECOUNT	NIL
MAKEDIRECTORY	NIL
WINDOWOPS	NIL
WINDOWDATA	NIL
CHECKFILENAME	NIL
HOSTALIVEP	NIL
OPENP	\GENERIC.OPENP
OPENFILELST	NIL
REGISTERFILE	\ADD-OPEN-STREAM

Pilot Floppy FDEV Functions

The following FLOPPY functions can be called by FILEIO

(\PFLOPPY.OPENFILE FILE ACCESS RECOG OTHERINFO FDEV OLDSTREAM)

Gets called when FILEIO opens a stream for input or output. If input, then \PFLOPPY.OPENOLDFILE eventually gets called. If output, then \PFLOPPY.OPENNEWFILE eventually gets called.

\PFLOPPY.OPENFILE returns a stream datatype. The DEVICE of the stream will be \PFLOPPYFDEV. Two other fields on the stream, F1 and F2, point to the allocation record (PFALLOC) and leader page (PLPAGE) for the stream. The PFALLOC and PLPAGE can be conveniently accessed by using the FLOPPYSTREAM ACCESSFNS. When other floppy functions are passed the stream to work with, FLOPPY looks at the arguments passed, the fields of the stream, and the fields of the PFALLOC and PLPAGE to determine how to act.

\PFLOPPY.DIR.GET is called to search for the allocation record of an old FILE. \PFLOPPY.DIR.PUT is called to store the allocation record of a newly created file.

(\PFLOPPY.READPAGES STREAM FIRSTPAGE# BUFFERS)

Reads sectors off floppy into virtual memory pages BUFFERS. FIRSTPAGE# is in FILEIO's scheme of counting the pages of a file, beginning with 0 for the first page of a file. \PFLOPPY.READPAGES therefore fills BUFFERS with data read from the floppy beginning with the FIRSTPAGE# of STREAM.

FIRSTPAGE# is in FILEIO's scheme of counting the pages of a file. FILEIO page numbers are converted into Pilot page numbers lieing somewhere between \PFLOPPYFIRSTDATAPAGE and \PFLOPPYLASTDATAPAGE. Pilot page numbers are in turn converted into DISKADDRESSes which record head, sector, and cylinder of a page on a floppy.

\PFLOPPY.READPAGES calls \FLOPPY.READPAGE which computes a Pilot page number from the stream's PFALLOC and the FILEIO page number passed in as an argument.

\FLOPPY.READPAGE calls \PFLOPPY.PAGENOTODISKADDRESS to convert the Pilot page number into a DISKADDRESS and then calls \FLOPPY.READSECTOR to read the sector at the computed disk address.

(\PFLOPPY.WRITEPAGES STREAM FIRSTPAGE# BUFFERS)

Other than that writing is taking place instead of reading, \PFLOPPY.WRITEPAGES is similar to \PFLOPPY.READPAGES.

Writes contents of virtual memory pages BUFFERS on to sectors of floppyfills BUFFERS with data . FIRSTPAGE# is in FILEIO's scheme of counting the pages of a file, beginning with 0 for the first page of a file. \PFLOPPY.WRITEPAGES therefore writes to the floppy beginning with the location corresponding to the FIRSTPAGE# of STREAM.

FIRSTPAGE# is in FILEIO's scheme of counting the pages of a file. FILEIO page numbers are converted into Pilot page numbers lieing somewhere between \PFLOPPYFIRSTDATAPAGE and \PFLOPPYLASTDATAPAGE. Pilot page numbers are in turn converted into DISKADDRESSes which record head, sector, and cylinder of a page on a floppy.

\PFLOPPY.WRITEPAGES calls \FLOPPY.WRITEPAGE which computes a Pilot page number from the stream's PFALLOC and the FILEIO page number passed in as an argument.

\FLOPPY.WRITEPAGE calls \PFLOPPY.PAGENOTODISKADDRESS to convert the Pilot page number into a DISKADDRESS and then calls \FLOPPY.WRITESECTOR to write the sector at the computed disk address.

(\PFLOPPY.TRUNCATEFILE FILE LASTPAGE LASTOFFSET)

Called just before closing an output file. The effect as far as FLOPPY is concerned is to take the allocation record (PFALLOC) for FILE and split the allocation record into two records if necessary. The first PFALLOC created this way is just big enough to store the truncated file. The second PFALLOC created this way becomes a free block. Since FLOPPY does not know how big an output file will turn out to be when it is first opened, FLOPPY must go through the process of allocating a reasonable size block, growing the block on occasion when \PFLOPPY.WRITEPAGES is about to cause the block to overflow, and finally truncate--i.e. split--the block into actual file and free block when \PFLOPPY.TRUNCATEFILE gets called.

\PFLOPPY.TRUNCATEFILE calls \PFLOPPY.TRUNCATE which does the actual splitting of a PFALLOC . \PFLOPPY.TRUNCATE splits a PFALLOC into two PFALLOCs equal to file and free block. The list of PFALLOCS cached on the floppy file device is updated and sufficiently many \PFLOPPY.WRITEPAGENOs of new or updated Pilot marker pages are written out to the floppy.

(\PFLOPPY.CLOSEFILE FILE)

Called to close a file. Writes out the leader page (PLPAGE) of FILE and the two marker pages that go around the leader page+ file.

(\PFLOPPY.DELETEFILE FILE FDEV)

Called to delete a file. \PFLOPPY.DIR.GET gets called to search for the allocation record for FILE. \PFLOPPY.DIR.REMOVE gets called to remove the allocation record from the cached directory alist. \PFLOPPY.DEALLOCATE gets called to turn the removed PFALLOC into a free block. The two marker pages surrounding FILE are updated to indicate that there is a free block between the marker pages and then they are written out. (Otherwise, at least until the free block gets used for other purposes, the contents of the file are still there, but have become inaccessible.)

(\PFLOPPY.RENAMEFILE OLDDEVICE OLDFILE NEWDEVICE NEWFILE OLDRECOG NEWRECOG)

Called to rename a file. \PFLOPPY.DIR.GET gets called to search for the allocation record for FILE. The PFALLOC found in the directory alist is removed with \PFLOPPY.DIR.REMOVE and then reentered with \PFLOPPY.DIR.PUT under the new file name which \PFLOPPY.RENAMEFILE computes. Finally, the leader page for the PFALLOC is changed to have the new file name and then is written out to the floppy.

(\PFLOPPY.GETFILEINFO FILE ATTRIBUTE FDEV)

Called by GETFILEINFO. All file attributes like WRITEDATE, CREATIONDATE, LENGTH, and TYPE are stored on the leader page of a file. \PFLOPPY.GETFILEINFO returns values, sometimes suitably converted, out of the leader page.

(\PFLOPPY.SETFILEINFO FILE ATTRIBUTE VALUE)

Called by SETFILEINFO. All file attributes like WRITEDATE, CREATIONDATE, LENGTH, and TYPE are stored on the leader page of a file. The VALUE for ATTRIBUTE, sometimes suitably converted, is stored into the leader page. The updated leader page is then written out to the floppy if FILE is not open. (If FILE is open then the leader page will be written out to the floppy when FILE is closed.)

(\PFLOPPY.GETFILENAME FILE RECOG FDEV)

Called by FINDFILE and INFILEP. This function can get called if {FLOPPY} (or a directory name like {FLOPPY}<MYDIR>) is on the user’s DIRECTORIES search path. When a function like OPENSTREAM fails to find a file on the user’s connected directory, \PFLOPPY.GETFILENAME may get asked about FILE to see if FILE is on {FLOPPY}.

\PFLOPPY.GETFILENAME is coded very similar to \PFLOPPY.DIR.GET which searches the floppy directory alist stored in (fetch (PFLOPPYFDEV DIR) of \FLOPPYFDEV).

\PFLOPPY.GETFILENAME returns NIL if no file is found. NIL is also returned if there is no floppy in the floppy drive or if there is no floppy drive on the machine that the user is using.

(\PFLOPPY.GENERATEFILES FDEV PATTERN DESIREDPROPS OPTIONS)

Called by DIRECTORY and the DIR LISPXMACRO. Like all corresponding GENERATEFILES functions installed on other file devices, \PFLOPPY.GENERATEFILES returns a file generator FILEGENOBJ which is a record that contains a GENFILESTATE plus two function names--in FLOPPY's case, \PFLOPPY.NEXTFILEFN and \PFLOPPY.FILEINFOFN.

The GENFILESTATE is a record that amounts to a stack of all the desired FLOPPY files according to the PATTERN, DESIREDPROPS, and OPTIONS that have been supplied. The files that go into the GENFILESTATE are collected by walking along the floppy directory alist which is stored on (fetch (PFLOPPYFDEV DIR) of \FLOPPYFDEV). A lot of the work is done by calling the function DIRECTORY.MATCH.

\PFLOPPY.NEXTFILEFN takes a GENFILESTATE, pops a file name out of the GENFILESTATE which also side effects the GENFILESTATE, and then returns the file name. If the GENFILESTATE has gone empty, then \PFLOPPY.NEXTFILEFN just returns NIL.

\PFLOPPY.FILEINFOFN is asked to supply information about the file that is at the top of the GENFILESTATE stack. This is done quite easily by calling the same function (\PFLOPPY.GETFILEINFO1) that \PFLOPPY.GETFILEINFO calls.

(\FLOPPY.HOSTNAMEP NAME FDEV)

This kind of function is intended for file devices that have nicknames. The HOSTNAMEP functions get called when the user specifies a file name containing a nickname as the host part of the file name. Since {FLOPPY} doesn't have any nicknames, \FLOPPY.HOSTNAMEP returns T iff NAME is FLOPPY.

(\FLOPPY.EVENTFN FDEV EVENT)

This function gets called before and after any LOGOUT, SYSOUT, MAKESYS, or SAVEVM.

Pilot Page Numbers

The hardware refers to locations of pages on a floppy by cylinder, head, and sector numbers. This way of enumerating pages on a floppy is cumbersome and the OSD Pilot design makes the somewhat sensible choice of making up its own numbering system for the pages on a floppy. We refer to these numbers as pilot page numbers.

\PFLOPPYFIRSTDATAPAGE (Variable)

\PFLOPPYLASTDATAPAGE (Variable)

Each cylinder+head+sector combination can be encoded as a single FIXP called a DISKADDRESS. The functions to convert between Pilot page numbers and DISKADDRESSes are

(\PFLOPPY.PAGENOTODISKADDRESS PAGENO)

(\PFLOPPY.DISKADDRESSTOPAGENO DISKADDRESS)

The functions to read and write pages to particular pilot page numbers are

(\PFLOPPY.READPAGENO PAGENO PAGE NOERROR)

(\PFLOPPY.WRITEPAGENO PAGENO PAGE NOERROR)

Pilot Marker Pages

Marker pages delimit the files and free blocks found on a Pilot floppy. The pattern of the data stored on a floppy is

DATA = MP ALLOC MP ALLOC MP ALLOC MP ... MP ALLOC MP

Generally a ALLOC is a free block or a leader page + file combination. One ALLOC is the filelist.

The declaration for Pilot floppy marker pages is

PMPAGE **(Datatype)**

```
(DATATYPE PMPAGE
  ((SEAL WORD)
   (VERSION WORD)
   (* Previous marker page entry *)
   (PLENGTH SWAPPEDFIXP)
   (PTYPE WORD)
   (PFILEID SWAPPEDFIXP)
   (PFILETYPE WORD)
   (NIL 121 WORD)
   (* Next marker page entry *)
   (NLENGTH SWAPPEDFIXP)
   (NTYPE WORD)
   (NFILEID SWAPPEDFIXP)
   (NFILETYPE WORD)
   (NIL 121 WORD)))
```

SEAL.PMPAGE **(Constant)**

VERSION.PMPAGE **(Constant)**

The SEAL and VERSION fields of a PMPAGE are the same for all marker pages. The magic constants are SEAL.PMPAGE and VERSION.PMPAGE. The SEAL.PMPAGE magic constant is arbitrary enough that there is only a slight chance that a non marker page would begin with this particular word. Hence, the scavenger can search for marker pages by looking for pages that begin with SEAL.PMPAGE.

The fields PLENGTH, PTYPE, PFILEID, PFILETYPE describe the ALLOC preceding the marker page. The fields NLENGTH, NTYPE, NFILEID, NFILETYPE describe the ALLOC following the marker page.

PLENGTH, NLENGTH = Length of ALLOC in pages.

PTYPE, NTYPE = Free, file, or filelist.

PFILEID, NFILEID = Pretty worthless. Used by Mesa. Each ALLOC has its own fileid number in the Mesa floppy handler. Not used by the Xerox Lisp software.

PFILETYPE, NFILETYPE =Free, file, or filelist. At first glance you might think that there isn't much point in having both PTYPE, NTYPE and PFILETYPE, NFILETYPE. If you think that, you are right. Please remember before you throw the rotten oranges that this is not my design. Blame it on OSD.

Pilot Leader Pages

The pattern of the data stored on a floppy is

DATA = MP ALLOC MP ALLOC MP ALLOC MP ... MP ALLOC MP

Generally an ALLOC is a free block or a leader page + file combination. One ALLOC is the filelist. Thus the possible patterns for an ALLOC look like

ALLOC = FREE
ALLOC = LP FILE
ALLOC = FILELIST

We note that all leader pages that occur on a Pilot floppy are immediately preceded by a marker page. All files on a Pilot floppy are preceded by a leader page. Each leader page + file combination is surrounded by two marker pages.

An exception to the rule: Microcode files on boot floppies do not have leader pages. They look like ALLOC = FILE. You may wonder why a microcode file's ALLOC shouldn't also be ALLOC = LP FILE. There isn't any reason for the exception. This is just another feature of the OSD design.

The declaration for Pilot floppy leader pages is

PLPAGE **(Datatype)**

```
(DATATYPE PLPAGE
 ((SEAL WORD)
  (VERSION WORD)
  (MESATYPE WORD)
  (* Offset 6 *)
  (\CREATIONDATE SWAPPEDFIXP)
  (\WRITEDATE SWAPPEDFIXP)
  (PAGELENGTH SWAPPEDFIXP)
  (HUGEPAGESTART SWAPPEDFIXP)
  (HUGEPAGELENGTH SWAPPEDFIXP)
  (HUGELENGTH SWAPPEDFIXP)
  (\NAMELENGTH WORD)
  (NAMEMAXLENGTH WORD)
  (* Offset 17 *)
  (\NAME 50 WORD)
  (* Offset 67 *)
  (UFO1 WORD)
  (UFO2 WORD)
  (DATAVERSION WORD)
  (\TYPE WORD)
  (NIL 183 WORD)
  (\BYTESIZE WORD))
```

The SEAL and VERSION fields of a PLPAGE are the same for all marker pages. The magic constants are SEAL.PLPAGE and VERSION.PLPAGE. The SEAL.PLPAGE magic constant is arbitrary enough that there is only a slight chance that a non marker page would begin with this particular word. Hence, the scavenger can search for marker pages by looking for pages that begin with SEAL.PLPAGE.

MESATYPE is fairly obscure. I think it marks the file according to what kind of application produced the file. Each application can register itself with OSD who doles out the numbers.

\CREATIONDATE is the obvious. However, the numeric encryption is according to Mesa's standards.

\WRITEDATE is the obvious. However, the numeric encryption is according to Mesa's standards.

PAGELENGTH is the length of the file after the leader page in pages.

HUGEPAGESTART, HUGEPAGELENGTH, HUGELENGTH are pretty arcane. For ordinairy files, HUGEPAGESTART = 0 and HUGEPAGELENGTH = PAGELENGTH. For all files, HUGELENGTH = the length of the file in bytes. For sysout and huge pilot files, it is possible that HUGEPAGESTART > 0 and HUGEPAGELENGTH < PAGELENGTH. Or some funny business like this.

\NAMELENGTH, NAMEMAXLENGTH, \NAME indicate the name of the file following the leader page. Oddly enough, NAMEMAXLENGTH is just a constant field for all leader pages.

UFO1, UFO2. Flying saucers. Treated as constant fields by Lisp.

DATAVERSION. Another constant field.

\TYPE = binary or text.

Pilot Sector 9

The name of the floppy is stored on the page at location CYLINDER = 0, HEAD = 0, SECTOR = 9. All of CYLINDER = 0 is wasted except for sector 9 . There is nothing valuable stored on the 8 sectors before sector 9, or any of the sectors on CYLINDER = 0.

All the tracks from CYLINDER=1 to \FLOPPY.CYLINDERS are formatted IBM double density 512 bytes per sector. Strangely, the two sides of CYLINDER = 0 are formatted differently. As part of the OSD design, CYLINDER = 0, HEAD = 0 is formatted IBM single density 128 bytes per sector and CYLINDER = 0, HEAD = 1 is formatted IBM double density 256 bytes per sector.

The pattern of the data stored in sector 9 is conveniently represented by a datatype called PSECTOR9.

PSECTOR9 **(Datatype)**

```
(DATATYPE PSECTOR9
  ((SEAL WORD)
   (VERSION WORD)
   (CYLINDERS WORD)
   (TRACKSPERCYLINDER WORD)
   (SECTORSPERTRACK WORD)
   (PFILELISTSTART WORD)
   (PFILELISTFILEID SWAPPEDFIXP)
   (PFILELISTLENGTH WORD)
```

```
(ROOTFILEID SWAPPEDFIXP)
(NIL WORD)
(PILOTMICROCODE WORD)
(DIAGNOSTICMICROCODE WORD)
(GERM WORD)
(PILOTBOOTFILE WORD)
(FIRSTALTERNATESECTOR WORD)
(COUNTBADSECTORS WORD)
(NEXTUNUSEDFILEID SWAPPEDFIXP)
(CHANGING FLAG)
(NIL BITS 15)
(\LABELLENGTH WORD)
(\LABEL 106 WORD)))
```

Lisp verifies that the SEAL of the PSECTOR9 that it reads in is equal to magic constant SEAL.PSECTOR9. This action serves to check that the floppy that is being read is in fact a Pilot floppy.

A floppy's name is stored in \LABELLENGTH and \LABEL. The floppy name is accessed by the user through the function FLOPPY.NAME.

Everything else about PSECTOR9 is not useful to Lisp and is not used by any of the Lisp code. However Mesa does use some of this cruft and Lisp has to keep the cruft consistent with what Mesa would like to see.

Occasionally, the filelist (also useless) moves around on the floppy. When this happens, PFILELISTSTART, PFILELISTFILEID, PFILELISTLENGTH also have to be updated.

NEXTUNUSEDFILEID has to do with the Mesa floppy handler's notion of what a fileid is. This notion is irrelevant to the Lisp implementation of FLOPPY, but the Lisp implementation of FLOPPY does have to keep NEXTUNUSEDFILEID equal to one plus the number of files currently stored on the floppy.

Every time a new file is created and closed, NEXTUNUSEDFILEID has to be increased by one, the filelist has to change, and both the PSECTOR9 and the filelist have to be written out to the floppy.

(\PFLOPPY.SAVE.PSECTOR9 NOERROR)

Saves the cached PSECTOR9 out to the floppy.

Pilot Filelist

The Pilot filelist is not used by the Lisp FLOPPY but must be maintained by the Lisp FLOPPY code to be compatible with the Mesa floppy handler. The filelist has the form

FILELIST = SEAL VERSION NENTRIES MAXENTRIES ENTRY
ENTRY ENTRY ... ENTRY

Every time a new file is created and closed, an ENTRY has to be added to FILELIST and FILELIST has to be written out. Every time a file is deleted, an ENTRY has to be deleted from FILELIST and FILELIST has to be written out. Other than to maintain the FILELIST in this way for Mesa's benefit, Lisp does not use the

FILELIST. Lisp uses the marker pages on a floppy to find where the files are and could care less about the filelist.

The filelist and filelist entry record declarations are

```
(BLOCKRECORD PFILELIST
  ((SEAL WORD)
    (VERSION WORD)
    (NENTRIES WORD)
    (MAXENTRIES WORD)))

(DATATYPE PFLE
  ((FILEID SWAPPEDFIXP)
    (TYPE WORD)
    (START WORD)
    (LENGTH WORD)))
```

(\PFLOPPY.ADD.TO.PFILELIST PFALLOC)

Create a PFLE file list entry on the filelist for this PFALLOC.

(\PFLOPPY.DELETE.FROM.PFILELIST PFALLOC)

Deletes the PFLE file list entry on the filelist corresponding to this PFALLOC.

(\PFLOPPY.SAVE.PFILELIST NOERROR)

Saves the filelist out to the floppy.

Pilot Floppy Format

The overall architecture of a Pilot floppy looks like this

FLOPPY = CYLINDER0 DATA

CYLINDER0 = CYLINDER0HEAD0 CYLINDER0HEAD1

CYLINDER0HEAD0 = GARBAGE SECTOR9 GARBAGE

CYLINDER0HEAD1 = GARBAGE

DATA = DATA = MP ALLOC MP ALLOC MP ... MP ALLOC MP

ALLOC = FREE
ALLOC = LP FILE
ALLOC = FILELIST

Exactly one ALLOC is the FILELIST. SECTOR9 and the FILELIST are for the most part useless. However, the Lisp implementation has to maintain SECTOR9 and FILELIST in a way that will make the Mesa floppy handler happy.

The Lisp implementation finds out what files are present on a floppy by following the marker pages (MPs) in the DATA which always begins at \PFLOPPYFIRSTDATAPAGE andends at \PFLOPPYLASTDATAPAGE. There are fields on the marker pages telling how long and what kind of ALLOC appears on either side of the marker page.

The formatting function for Pilot floppies is

(\PFLOPPY.FORMAT NAME AUTOCONFIRMFLG SLOWFLG)

\PFLOPPY.FORMAT formats the tracks of the floppy according to the peculiar OSD design and then makes the floppy look like it is a Pilot floppy with an empty directory. The DATA on a freshly formatted floppy has the form

DATA = MP FILELIST MP FREE MP

Cached Pilot Floppy Information

Duplicate directory information about a floppy is cached in core. This is held on to by a PFINFO datatype.

PFINFO **(Datatype)**

(DATATYPE PFINFO (OPEN PFILELIST PFALLOCS DIR PSECTOR9))

We may describe the fields of the PFINFO slightly out of order.

OPEN is a flag saying whether directory information for the current floppy has been cached or not in th remaining fields of the PFINFO.

PFILELIST is the Pilot filelist. The filelist is unimportant except that it must be maintained for Mesa's benefit. An updated PFILELIST has to be written out each time a file is written etc.

PSECTOR9 is the Pilot sector 9 record. This is where the floppy name lives. Otherwise, this record is also pretty worthless as far as Lisp is concerned. But there are fields on the PSECTOR9 that have to be maintained for Mesa's benefit. An updated PFILELIST has to be written out each time a file is written etc.

DIR is the cached flopy directory alist which is an alist of alists of alists.

PFALLOCS is a list of PFALLOC records which describe successive allocations on the floppy.

Cached Pilot Floppy DIR Alist

The PFINFO DIR is the cached flopy directory alist which is an alist of alists of alists. It has the form

```
((name (extension (version . PFALLOC) ...
                              (version . PFALLOC)) ...
          (extension (version . PFALLOC) ...
                              (version . PFALLOC))) ...
 (name (extension (version . PFALLOC) ...
                              (version . PFALLOC)) ...
          (extension (version . PFALLOC) ...
                              (version . PFALLOC))))
```

Say for example that we have 6 versions of the file {FLOPPY}FLOPPY.TEDIT stored on a floppy. Then the cached DIR alist could look like

```
((FLOPPY (TEDIT (1 . {PFALLOC}#55,72740)
                (2 . {PFALLOC}#55,72704)
                (3 . {PFALLOC}#55,72650)
                (4 . {PFALLOC}#55,72434)
                (5 . {PFALLOC}#55,72400)
                (6 . {PFALLOC}#55,72340))))
```

Functions to manage and access the DIR alist are

(\PFLOPPY.DIR.GET FILENAME RECOG)

(\PFLOPPY.DIR.PUT FILENAME RECOG PFALLOC)

(\PFLOPPY.DIR.REMOVE PFALLOC)

(\PFLOPPY.DIR.VERSION VERSION RECOG VALIST FILENAME)

Cached Pilot Floppy PFALLOCS

The PFINFO PFALLOCS is a list of PFALLOC records which describe successive allocations on the floppy. Some of the PFALLOCs are leader page+file combinations. Some are free blocks. One of the PFALLOCs is the allocation record for the filelist (this PFALLOC tells us where the contents of PFILELIST should be stored).

Each PFALLOC record has the form

PFALLOC **(Datatype)**

```
(DATATYPE PFALLOC
  (FILENAME
    (PREV FULLXPOINTER)
    NEXT
    START
    PMPAGE
    PLPAGE
    PFLE
    (WRITEFLG FLAG)
    (DELETEFLG FLAG)))
```

FILENAME is the name of the file that the PFALLOC corresponds to. If the PFALLOC does not correspond to a file, then FILENAME will be a list like (FREE) or (PFILELIST).

PREV points back to the preceding PFALLOC in the PFALLOCS list.

NEXT points to the next PFALLOC in the PFALLOCS list.

START is the pilot page number of the first page of storage on the floppy corresponding to the PFALLOC (the page after the marker page preceding the allocation).

PMPAGE is a cached copy of the marker page preceding the allocation corresponding to the PFALLOC.

PLPAGE is a cached copy of the leader page, if any, that begins the allocation. (Only file allocations will have leader pages. If the PFALLOC does not correspond to a file allocation then the PLPAGE field is left NIL.)

PFLE is the file list entry in the Pilot filelist corresponding to this PFALLOC.

WRITEFLG = T if there is a stream writing to the allocation corresponding to this PFALLOC. That is, this PFALLOC corresponds to a file, and a stream is writing to that file.

DELETEFLG = T if the PFALLOC corresponds to a file whose deletion is pending. This may happen if some process is reading a file that a second process DELFILEs. The PFALLOC is deleted for real (i.e. made into a free block) when the first process gives up control of the stream that is reading the file with CLOSEF.

PILOT FLOPPY STORAGE ALLOCATION

This section describes the Pilot floppy storage allocation strategy.

(\PFLOPPY.ALLOCATE LENGTH)

This function is called to generate the initial allocation PFALLOC assigned to a stream. The PFALLOC must be of a definite length and so there are other functions like \PFLOPPY.TRUNCATE and \PFLOPPY.EXTEND which know how to split an incompletely used up PFALLOC or a PFALLOC that is about to overflow.

\PFLOPPY.ALLOCATE returns a PFALLOC pointing to a free block of storage that is at least LENGTH pages long. If LENGTH=NIL is supplied, then the length is defaulted to somewhere between DEFAULT.ALLOCATION and MINIMUM.ALLOCATION with preference towards DEFAULT.ALLOCATION.

The strategy is to first select the largest free block available. Not being able to find a free block causes \PFLOPPY.GAINSPACE to be called and then \PFLOPPY.ALLOCATE retries.

Having obtained the PFALLOC pointing to the largest free block available, \PFLOPPY.ALLOCATE determines if the free block is big enough. If the free block is not big enough then \PFLOPPY.GAINSPACE is called and \PFLOPPY.ALLOCATE retries.

Now being in possession of a PFALLOC pointing to a sufficiently large free block, the question is whether or not to split the PFALLOC. If the length of PFALLOC exceeds LENGTH by MINIMUM.ALLOCATION then PFALLOC is split into two free blocks one of which will have length LENGTH and will be returned. Otherwise, the PFALLOC is returned as is.

Splitting excessively large PFALLOCs is done in order that no stream hog the floppy and thereby be unfair to other streams. It may happen, for example in a freshly formatted floppy, that there is just one or very few but large free blocks. It would be unfair to hand out the only and very large free block to a stream and thereby prevent a second stream from opening.

On the other hand, splitting a PFALLOC whose length exceeds LENGTH by just a little bit is also undesirable because it leads to fragmentation and consumption of storage by the marker page overhead that must occur for each allocation. Thus, the length of PFALLOC not exceeding LENGTH by MINIMUM.ALLOCATION is considered to be close enough.

\PFLOPPY.ICHECK is called at the end of each \PFLOPPY.ALLOCATE to do an integrity check of the cached incore description of the floppy.

(\PFLOPPY.DEALLOCATE PFALLOC)

Deallocation is fairly easy. The two marker pages surrounding the allocation pointed to by PFALLOC are made to say that the type of the allocation between them is a free block.

\PFLOPPY.ICHECK is called at the end of each \PFLOPPY.DEALLOCATE to do an integrity check of the cached incore description of the floppy.

(\PFLOPPY.TRUNCATE PFALLOC LENGTH)

Truncation is also fairly easy. If PFALLOC is already of length LENGTH or smaller then nothing needs to be done.

Otherwise \PFLOPPY.TRUNCATE changes the local situation on the floppy from one of

MP ALLOC MP

to one of

MP ALLOC MP FREE MP

by setting the length of PFALLOC to LENGTH, creating another PFALLOC to take up the slack and to be considered as a free block, and creating and updating and writing out three marker pages.

\PFLOPPY.ICHECK is called at the end of each \PFLOPPY.TRUNCATE to do an integrity check of the cached incore description of the floppy.

(\PFLOPPY.EXTEND PFALLOC)

This function gets called when an output stream is about to overflow its initial allocation. \PFLOPPY.EXTEND is charged with extending the length of PFALLOC's storage allocation.

If PFALLOC is followed by a free block, then it suffices to let PFALLOC cannibalize the following free block. The situation

MP ALLOC MP FREE MP

is changed to one of

MP ALLOC MP

by changing the length of PFALLOC to the sum of the length of PFALLOC’s length, one page from the marker page between PFALLOC and the free block which is eliminated, and the length of the free block.

If PFALLOC is not followed by a free block, but instead is bumping up against valuable data, then a different strategy is used. \PFLOPPY.ALLOCATE is called to find a new and larger storage area NEW for the data stored at the allocation pointed to by PFALLOC. The data stored in the allocation pointed to by PFALLOC is then copied into the allocation pointed to by NEW. This is slightly time consuming, so the Lisp code prints the message "Reallocating" in the PROMPTWINDOW when this kind of activity is about to happen. After copying the data pointed to by PFALLOC into the area pointed to by NEW, NEW nad PFALLOC are made to exchange places. NEW is changed to point to where the data used to live and PFALLOC is changed to point to where the data has moved to. The marker pages around NEW and around PFALLOC have to be updated and written out to the floppy. The result, therefore, has the appearance of data crawling out of its restricted cavity and kicking out the free block living from the bigger cavity and then the free block moves over to where the data used to live.

\PFLOPPY.ICHECK is called at the end of each \PFLOPPY.EXTEND to do an integrity check of the cached incore description of the floppy.

(\PFLOPPY.GAINSPACE LENGTH)

This function is charged with going out and hunting up space on a floppy. \PFLOPPY.GAINSPACE returns after a free block of length LENGTH has been made available.

The first thing to be done is to merge adjacent free blocks into larger free blocks. This is done by calling function \PFLOPPY.GAINSPACE.MERGE. \PFLOPPY.ICHECK is called at the end of each \PFLOPPY.GAINSPACE.MERGE to do an integrity check of the cached incore description of the floppy.

At this point \PFLOPPY.GAINSPACE checks to see if a free block of length LENGTH has been made available, and if so, returns.

Otherwise, \PFLOPPY.GAINSPACE calls FLOPPY.FREE.PAGES to determine whether a sufficiently large free block can be gained just by compacting the floppy. If the number of free pages available is greater than LENGTH, then compacting the floppy will have as one of its effects the collection of all free space into a single free block. Therefore, if the number of free pages available is greater than LENGTH, then \PFLOPPY.GAINSPACE calls FLOPPY.COMPACT to compact the floppy and then returns the single free block that is created by the compactor.

If calling FLOPPY.FREE.PAGES tells \PFLOPPY.GAINSPACE that there wouldn’t be a large enough free block created just by compacting the floppy, then \PFLOPPY.GAINSPACE generates the usual contiuable FILE SYSTEM RESOURCES EXCEEDED error break. If the user responds by deleting some files and typing OK, then \PFLOPPY.GAINSPACE continues onward by going back to the top of its list of things to do and retrying its attempt to find a free block of length LENGTH.

(\PFLOPPY.FREE.PAGES)

Just sums up the lengths of all the PFALLOCs pointing to free blocks. This function gets called by the user function FLOPPY.FREE.PAGES.

(\PFLOPPY.ICHECK)

Does an integrity check on the Lisp implementation’s incore cached directory information.

The Lisp floppy implementation does not blithely trust itself to always be doing the right thing, at least as far as the incore description of how the floppy is arranged. It is somewhat important that the in core description be absolutely legal and in agreement with reality out on the floppy, because incorrect cached directory structures may cause floppy operations to scramble the user’s floppy confusing contents of files or real directory structure on the floppy.

\PFLOPPY.ICHECK gets called near the end of each major function in the Pilot floppy allocation code.

SYSOUT

Sysouting to floppy is kind of a hack. The sysout file that gets written has to be broken into smaller files that will fit on individual floppies. The small files are not ordinairy files but are Huge Pilot files. The HUGEPAGESTART and HUGEPAGELENGTH fields in the leader pages of these files become important.

\SFLOPPYFDEV (Variable)

The usual FLOPPY file device is \PFLOPPYFDEV. The sysout FLOPPY file device is \SFLOPPYFDEV. (FLOPPY.MODE ’SYSOUT) by the user makes the switch between the two file devices. The function SYSOUT seems to do (FLOPPY.MODE ’SYSOUT) automatically for the user, but in fact, floppy functions at a lower level detect SYSOUT on the stack of function calls by a certain amount of grungeyness and make the switch between file devices when necessary.

(\SFLOPPY.OPENHUGEFILE FILE ACCESS RECOG OTHERINFO FDEV OLDSTREAM)

Installed on \SFLOPPYFDEV and gets called when FILEIO opens a stream for input or output when FLOPPY is in SYSOUT mode. \SFLOPPY.OPENHUGEFILE returns a stream datatype. The stream can be either input or output.

If the stream is to be an input stream, \SFLOPPY.INPUTFLOPPY is called. If the stream is to be an output stream, \SFLOPPY.OUTPUTFLOPPY is called.

The stream returned is in every way like a Pilot stream returned by \PFLOPPY.OPENFILE with the exception that the DEVICE of the stream is \SFLOPPYFDEV. Two other fields on the stream, F1 and

F2, point to the allocation record (PFALLOC) and leader page (PLPAGE) for the stream. The PFALLOC and PLPAGE can be conveniently accessed by using the FLOPPYSTREAM ACCESSFNS.

(\SFLOPPY.READPAGES STREAM FIRSTPAGE# BUFFERS)

Installed on \SFLOPPYFDEV and called by FILEIO when FLOPPY is in SYSOUT mode. Reads sectors off floppies into virtual memory pages BUFFERS. FIRSTPAGE# is in FILEIO's scheme of counting the pages of a file, beginning with 0 for the first page of a file. \SFLOPPY.READPAGES therefore fills BUFFERS with data read from floppies beginning with the FIRSTPAGE# of STREAM.

\SFLOPPY.READPAGES is implemented by calling \PFLOPPY.READPAGE. When \SFLOPPY.READPAGES is about to run off the end of a floppy, \SFLOPPY.CLOSEFLOPPY and \SFLOPPY.INPUTFLOPPY are called to bring in the next floppy.

(\SFLOPPY.WRITEPAGES STREAM FIRSTPAGE# BUFFERS)

Installed on \SFLOPPYFDEV and called by FILEIO when FLOPPY is in SYSOUT mode. Writes contents of virtual memory pages BUFFERS on to sectors of floppies. FIRSTPAGE# is in FILEIO's scheme of counting the pages of a file, beginning with 0 for the first page of a file. \SFLOPPY.WRITEPAGES therefore writes to the floppy beginning with the location corresponding to the FIRSTPAGE# of STREAM.

\SFLOPPY.WRITEPAGES is implemented by calling \PFLOPPY.WRITEPAGE. When \SFLOPPY.WRITEPAGES is about to run off the end of a floppy, \SFLOPPY.CLOSEFLOPPY and \SFLOPPY.OUTPUTFLOPPY are called to bring in the next floppy.

(\SFLOPPY.CLOSEHUGEFILE STREAM)

Installed on \SFLOPPYFDEV and called by FILEIO to close a sysout file when FLOPPY is in SYSOUT mode.

For output streams, does the usual \CLEARMAP plus a \SFLOPPY.CLOSEFLOPPY on the last floppy being written.

There is some grungeyness to switch back to an old FLOPPY mode if the FLOPPY mode was reset by calling function SYSOUT.

(\SFLOPPY.INPUTFLOPPY FLOPPYNAME FILENAME OTHERINFO OLDSTREAM)

Called by \SFLOPPY.READPAGES when \SFLOPPY.READPAGES needs to go to the next floppy. \SFLOPPY.INPUTFLOPPY prompts the user to "Insert floppy" and does a (FLOPPY.WAIT.FOR.FLOPPY T) until the user does so.

Once the user has inputted the next floppy and FLOPPY.WAIT.FOR.FLOPPY has returned, \SFLOPPY.INPUTFLOPPY calls \PFLOPPY.OPENFILE to open a regular Pilot floppy stream to the piece of sysout file stored on the floppy inserted.

If no OLDSTREAM is supplied, then the Pilot floppy stream created by the call to \PFLOPPY.OPENFILE is returned. Otherwise, the PFALLOC and PLPAGE cached in the F1 and F2 fields of the Pilot floppy stream are pulled out and stuck into OLDSTREAM, after which OLDSTREAM is returned.

(\SFLOPPY.OUTPUTFLOPPY FLOPPYNAME FILENAME OTHERINFO OLDSTREAM)

Called by \SFLOPPY.WRITEPAGES when \SFLOPPY.WRITEPAGES needs to go to the next floppy. \SFLOPPY.OUTPUTFLOPPY prompts the user to "Insert floppy" and does a (FLOPPY.WAIT.FOR.FLOPPY T) until the user does so.

Once the user has inputted the next floppy and FLOPPY.WAIT.FOR.FLOPPY has returned, \SFLOPPY.OUTPUTFLOPPY tries to format the floppy. If there is any problem with formatting the floppy, such as the floppy being writeprotected, then the user is again asked to input a floppy.

Once the new floppy is formatted, \SFLOPPY.OUTPUTFLOPPY calls \PFLOPPY.OPENFILE to open a regular Pilot floppy stream which will be used to create a piece of the sysout file being stored on floppies.

If no OLDSTREAM is supplied, then the Pilot floppy stream created by the call to \PFLOPPY.OPENFILE is returned. Otherwise, the PFALLOC and PLPAGE cached in the F1 and F2 fields of the Pilot floppy stream are pulled out and stuck into OLDSTREAM, after which OLDSTREAM is returned.

(\SFLOPPY.CLOSEFLOPPY STREAM LASTFLOPPYFLG)

If the STREAM is an input stream, then \SFLOPPY.CLOSEFLOPPY just returns without doing anything.

If the STREAM is an output stream, then the leader pages and marker pages for the piece of sysout file stored on this floppy are written out. And as usual with Pilot streams, the updated filelist and PSECTOR9 also have to be written out.

HUGEPILOT

HUGEPILOT mode is implemented in a way similar to SYSOUT mode. The Huge Pilot file that gets written has to be broken into smaller files that will fit on individual floppies. The small files are not ordinairy files but are Huge Pilot files. The HUGEPAGESTART and HUGEPAGELENGTH fields in the leader pages of these files become important.

\HFLOPPYFDEV (Variable)

The usual FLOPPY file device is \PFLOPPYFDEV. The Huge Pilot FLOPPY file device is \HFLOPPYFDEV. (FLOPPY.MODE 'HUGEPILOT) by the user makes the switch between the two file devices.

(\HFLOPPY.OPENHUGEFILE FILE ACCESS RECOG OTHERINFO FDEV OLDSTREAM)

Installed on \HFLOPPYFDEV and gets called when FILEIO opens a stream for input or output when FLOPPY is in HUGEPILOT mode.

\HFLOPPY.OPENHUGEFILE returns a stream datatype. The stream can be either input or output.

If the stream is to be an input stream, \HFLOPPY.INPUTFLOPPY is called. If the stream is to be an output stream, \HFLOPPY.OUTPUTFLOPPY is called.

The stream returned is in every way like a Pilot stream returned by \PFLOPPY.OPENFILE with the exception that the DEVICE of the stream is \HFLOPPYFDEV. Two other fields on the stream, F1 and F2, point to the allocation record (PFALLOC) and leader page (PLPAGE) for the stream. The PFALLOC and PLPAGE can be conveniently accessed by using the FLOPPYSTREAM ACCESSFNS.

(\HFLOPPY.READPAGES STREAM FIRSTPAGE# BUFFERS)

Installed on \HFLOPPYFDEV and called by FILEIO when FLOPPY is in HUGEPILOT mode. Reads sectors off floppies into virtual memory pages BUFFERS. FIRSTPAGE# is in FILEIO's scheme of counting the pages of a file, beginning with 0 for the first page of a file. \HFLOPPY.READPAGES therefore fills BUFFERS with data read from floppies beginning with the FIRSTPAGE# of STREAM.

\HFLOPPY.READPAGES is implemented by calling \PFLOPPY.READPAGE. When \HFLOPPY.READPAGES is about to run off the end of a floppy, \HFLOPPY.CLOSEFLOPPY and \HFLOPPY.INPUTFLOPPY are called to bring in the next floppy.

(\HFLOPPY.WRITEPAGES STREAM FIRSTPAGE# BUFFERS)

Installed on \HFLOPPYFDEV and called by FILEIO when FLOPPY is in HUGEPILOT mode. Writes contents of virtual memory pages BUFFERS on to sectors of floppies. FIRSTPAGE# is in FILEIO's scheme of counting the pages of a file, beginning with 0 for the first page of a file. \HFLOPPY.WRITEPAGES therefore writes to the floppy beginning with the location corresponding to the FIRSTPAGE# of STREAM.

\HFLOPPY.WRITEPAGES is implemented by calling \PFLOPPY.WRITEPAGE. When \HFLOPPY.WRITEPAGES is about to run off the end of a floppy, \HFLOPPY.CLOSEFLOPPY and \HFLOPPY.OUTPUTFLOPPY are called to bring in the next floppy.

(\HFLOPPY.CLOSEHUGEFILE STREAM)

Installed on \HFLOPPYFDEV and called by FILEIO to close a Huge Pilot file when FLOPPY is in HUGEPILOT mode.

For output streams, does the usual \CLEARMAP plus a \HFLOPPY.CLOSEFLOPPY on the last floppy being written.

(\HFLOPPY.INPUTFLOPPY FLOPPYNAME FILENAME OTHERINFO OLDSTREAM)

Called by \HFLOPPY.READPAGES when \HFLOPPY.READPAGES needs to go to the next floppy. \HFLOPPY.INPUTFLOPPY prompts the user to "Insert floppy" and does a (FLOPPY.WAIT.FOR.FLOPPY T) until the user does so.

Once the user has inputted the next floppy and FLOPPY.WAIT.FOR.FLOPPY has returned,

\HFLOPPY.INPUTFLOPPY calls \PFLOPPY.OPENFILE to open a regular Pilot floppy stream to the piece of Huge Pilot file stored on the floppy inserted.

If no OLDSTREAM is supplied, then the Pilot floppy stream created by the call to \PFLOPPY.OPENFILE is returned. Otherwise, the PFALLOC and PLPAGE cached in the F1 and F2 fields of the Pilot floppy stream are pulled out and stuck into OLDSTREAM, after which OLDSTREAM is returned.

(\HFLOPPY.OUTPUTFLOPPY FLOPPYNAME FILENAME OTHERINFO OLDSTREAM)

Called by \HFLOPPY.WRITEPAGES when \HFLOPPY.WRITEPAGES needs to go to the next floppy. \HFLOPPY.OUTPUTFLOPPY prompts the user to "Insert floppy" and does a (FLOPPY.WAIT.FOR.FLOPPY T) until the user does so.

Once the user has inputted the next floppy and FLOPPY.WAIT.FOR.FLOPPY has returned, \HFLOPPY.OUTPUTFLOPPY tries to format the floppy. If there is any problem with formatting the floppy, such as the floppy being writeprotected, then the user is again asked to input a floppy.

Once the new floppy is formatted, \HFLOPPY.OUTPUTFLOPPY calls \PFLOPPY.OPENFILE to open a regular Pilot floppy stream which will be used to create a piece of the Huge Pilot file being stored on floppies.

If no OLDSTREAM is supplied, then the Pilot floppy stream created by the call to \PFLOPPY.OPENFILE is returned. Otherwise, the PFALLOC and PLPAGE cached in the F1 and F2 fields of the Pilot floppy stream are pulled out and stuck into OLDSTREAM, after which OLDSTREAM is returned.

(\HFLOPPY.CLOSEFLOPPY STREAM LASTFLOPPYFLG)

If the STREAM is an input stream, then \HFLOPPY.CLOSEFLOPPY just returns without doing anything.

If the STREAM is an output stream, then the leader pages and marker pages for the piece of Huge Pilot file stored on this floppy are written out. And as usual with Pilot streams, the updated filelist and PSECTOR9 also have to be written out.

Font/Character Documentation

Greg Nuyens

filed as: {eris}<lispcore>internal>doc>font&chars.tedit
last edited: March 24, 1986

Notice:

this is a draft made available for comments. Please do not forward copies.

Comments are encouraged. Please send them to Nuyens.pa@Xerox.com

---

This note provides information about font and character facilities in Interlisp-D. It is organised in two parts:

1) a user level view of the changes involved in including NS characters in Interlisp-D (adapted from the Koto release notes ({Erinyes}<doc>koto>releasenotes>*))
2) a description of the underlying data-structures and facilities. (exported macros and fns, etc)

User-level view

Interlisp-D now supports the Xerox corporate character code standard, commonly referred to as the NS (Network Systems) encoding, described in the document *Character Code Standard* [Xerox System Integration Standards, XSIS 058404, April 1984]. Previous to the Koto release, Interlisp-D used the ASCII (American Standard Code for Information Interchange) encoding. While the extended-ASCII encoding provided for 8-bit (256 available) characters (primarily Latin alphabet and computer-specific symbols), the NS encoding supports 16-bit (65536 available) characters comprising many foreign alphabets and special symbols.

The benefit of having this large character set, in contrast to approaches that use a small set of character codes and a multiplicity of fonts (e.g., a Greek font, a math font), is that each semantically distinct character is represented by its own character code, completely independent of the character's appearance (font). Thus, the Greek character upper-case Beta is always character code 9794, independent of whether it appears in printed form in a serif style, sans-serif style, italic, etc., and it is unrelated to the Roman letter B (character code 66).

NS characters can be used in strings, litatom print names, symbolic files, or anywhere else that characters can be used. All of the standard string and print name functions (RPLSTRING, GNC, NCHARS, STRPOS, etc.) accept litatoms and strings containing NS characters. For example:

```
_(STRPOS "char" "this is an 8-bit character string")

18

_(STRPOS "char" "celui-ci comporte des charactères NS")

23
```

Characters are organized into 256-member *character sets*, each of which generally consists of semantically related characters. For example, character set 38 is the Greek character set and contains the Greek alphabet and punctuation characters needed to print Greek text. A 16-bit character code thus consists of an 8-bit character set and an 8-bit character number within that set. The ASCII character set is contained in NS character set zero; thus, ASCII characters are still represented by the same 8-bit character codes as previously (i.e., 16-bit character codes whose high 8 bits are zero). Most strings and atoms still consist entirely of characters from character set zero and are represented just as space-efficiently in memory and on files as in earlier releases of Interlisp-D that used only ASCII characters.

In almost all cases, a program does not need to know that it is dealing with 16-bit characters rather than 8-bit characters—the higher level system functions all treat them transparently. The exception is in character-level input/output, where the important fact to be aware of is that *characters are not bytes*. The file pointer of a random-access file still counts bytes, and the function NCHARS still counts characters, but the two are no longer directly related. This is discussed in more detail below.

Character-level Input/Output

Incompatible Change: **BIN and BOUT are no longer appropriate for character input/output.**

A character is no longer generally representable in 8 bits. Therefore, characters can no longer, in general, be read or written with the functions BIN and BOUT, which read and write 8-bit quantities. The change is mostly transparent to user programs, especially if those programs use only the higher level functions, such as READ and PRINT. However, it is likely that user programs that manipulated a file character by character using BIN and BOUT should now use the following functions, which may produce or consume more than a single byte:

(READCCODE *STREAM*) [Function]

(PEEKCCODE *STREAM*) [Function]

(PRINTCCODE *CODE STREAM*) [Function]

These functions are documented in the new Interlisp-D Reference Manual. The functions BIN and BOUT are still appropriate for use when reading and writing strictly binary (rather than character) data.

Interlisp-D supports two ways of writing NS characters on files. One way is to write the full 16-bits (two bytes) every time a character is output. The other way, which is the system default, is to use "run-encoding," in which a run of characters in the same character set is written as a sequence of 8-bit character numbers within the character set, preceded by a "change character set" command. The byte 255 (illegal as either a character set number or a character number) followed by a character set number is used to signal a change to a given character set; the following bytes, up until the next change-character set sequence, are all interpreted as coming from the specified character set. Run-encoding can reduce the number of bytes required to encode a string of NS characters, as long as there are long sequences of characters from the same character set, which is usually the case.

Most characters in common use, including those in the ASCII character set, are in character set zero; a file containing only these characters is thus in exactly the same format as in previous releases, viz., one byte per character. However, this should not be relied on.

The fact that the file representation of a character may be more than a single byte has important consequences for any program that uses random access on text files whose characters are run-encoded. First, and most obviously, you cannot count the characters in a string being printed and use that number to derive the file pointer of where the string ends—you must use GETFILEPTR. Second, programs that use SETFILEPTR need to be aware of possible character set changes. At any point when a file is being read or written, it has a "current character set," viz., the character set specified in the most recent "change character set" command written on the file. If the file pointer is changed with SETFILEPTR to a part of the file with a different character set, any characters read or written may have the wrong character set. Programs that use COPYBYTES to copy blocks of characters must ensure both that they are copying on character boundaries and copying to a place that is in the correct character set.

(Internal Note: PRINTCCODE is the user entry to the OUTCHARFN of the stream. It is bounds checked.)

The current character set can be accessed with the following function:

(CHARSET *STREAM CHARACTERSET*) [Function]

Returns the current character set of the stream *STREAM*, or T if *STREAM* is not run-encoded. If *CHARACTERSET* is non-NIL, the current character set for *STREAM* is set. For output streams this causes bytes to be written to the stream if *CHARACTERSET* is different from the current character set; for input streams it merely changes the reader's belief about the current character set. If *CHARACTERSET* is T, run encoding for *STREAM* is disabled—henceforth each character printed to the stream is printed as exactly two bytes (the character set and the character number).

Programs that wish to count characters or avoid worrying about character set changes can thus disable run encoding for a particular stream and count each character as two bytes. There is, however, a cost in file space.

Extensions to CHARCODE

CHARCODE has been extended to allow specifying NS Characters.

CHARCODE has been extended to allow the specification of 16-bit NS characters in multiple character sets. It also uses two new variables, CHARACTERNAMES and CHARACTERSETNAMES, so characters and character sets can be specified symbolically. The new definition is the following:

(CHARCODE *CHAR*) [NLambda Function]

Returns the character code specified by *CHAR* (unevaluated). If *CHAR* is a one-character atom or string, the corresponding character code is simply returned. Thus, (CHARCODE A) is 65, (CHARCODE 0) is 48. If *CHAR* is a multi-character litatom or string, it specifies a character code as described below. If *CHAR* is NIL, CHARCODE simply returns NIL. Finally, if *CHAR* is a list structure, the value is a copy of *CHAR* with all the leaves replaced by the corresponding character codes. For instance, (CHARCODE (A (B C))) => (65 (66 67)).

If a character is specified by a multi-character litatom or string, CHARCODE interprets it as follows:

CR, SPACE, etc. The variable CHARACTERNAMES contains an association list mapping special litatoms to character codes. Among the characters defined this way are CR (13), LF (10), SPACE or SP (32), ESCAPE or ESC (27), BELL (7), BS (8), TAB (9), NULL (0), and DEL (127). Examples: (CHARCODE SPACE) returns 32, and (CHARCODE CR) returns 13.

CHARSET,CHARNUM, CHARSET-CHARNUM If the character specification is a litatom or string of the form CHARSET,CHARNUM or CHARSET-CHARNUM, the character code for the character number CHARNUM in the character set CHARSET is returned. CHARSET is either an octal number, or a litatom in the association list CHARACTERSETNAMES (which defines GREEK, CYRILLIC, etc.). CHARNUM is either an octal number, a single-character litatom, or a litatom from the association list CHARACTERNAMES. Examples: (CHARCODE 12,6), (CHARCODE 12,SPACE), (CHARCODE GREEK,A) and (CHARCODE ^GREEK,A)

Note that if CHARNUM is a single-digit number, it is interpreted as an octal character code, *not* as a character. Thus (CHARCODE GREEK,3) denotes the fourth character in the Greek character set, not the character "3" in that character set.

^CHARSPEC If the character specification is a litatom or string of one of the forms above, preceded by the character "^", this indicates a "control character," derived from the normal character code by clearing the seventh bit (100Q) of the character code (normally set in alphabetic characters). Example: (CHARCODE ^A)

#CHARSPEC (8-bit character codes) If the character specification is a litatom or string of one of the forms above, preceded by the character "#", the eighth bit (200Q), normally zero for 7-bit ASCII characters, is set. This is the way to get character numbers greater than 127. ^ and # can both be set at once. Examples: (CHARCODE #A), (CHARCODE #^GREEK,A)

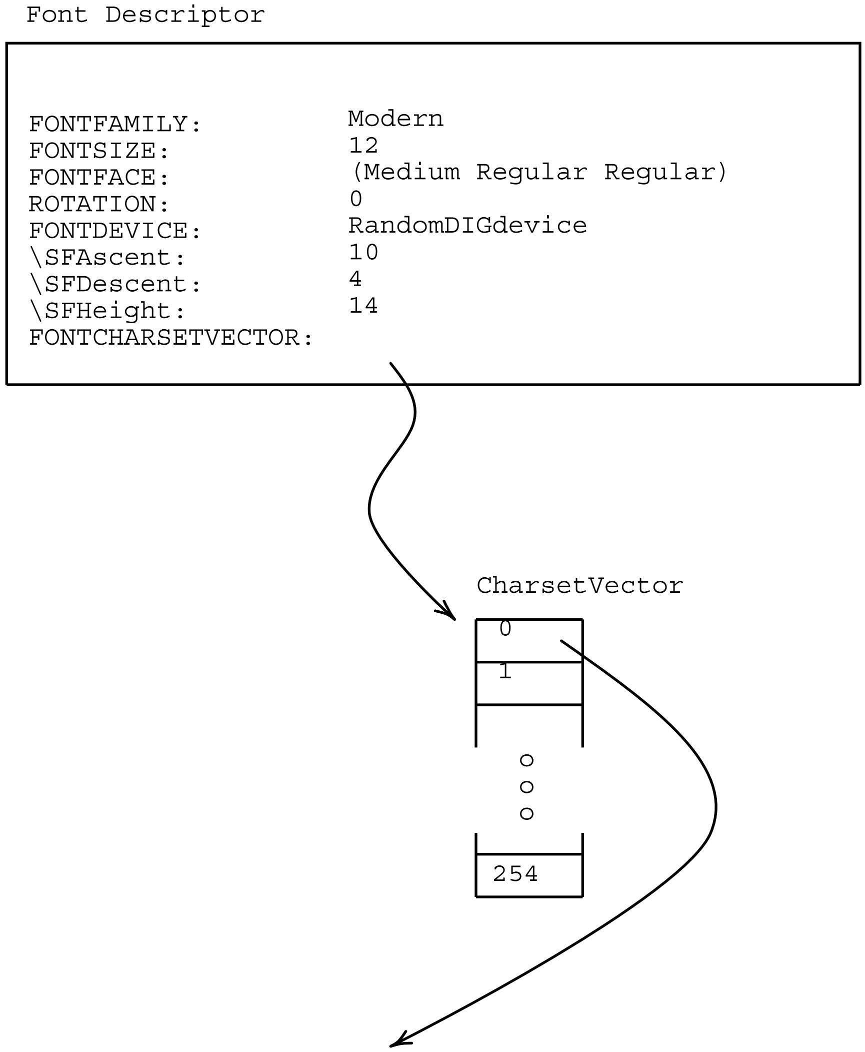
Note: In Intermezzo (and in some other operating systems), characters with the eighth bit set were considered "meta" characters. In the Koto release, however, "meta" means character set 1, and the meta key produces characters with the 400Q bit set, not 200Q.

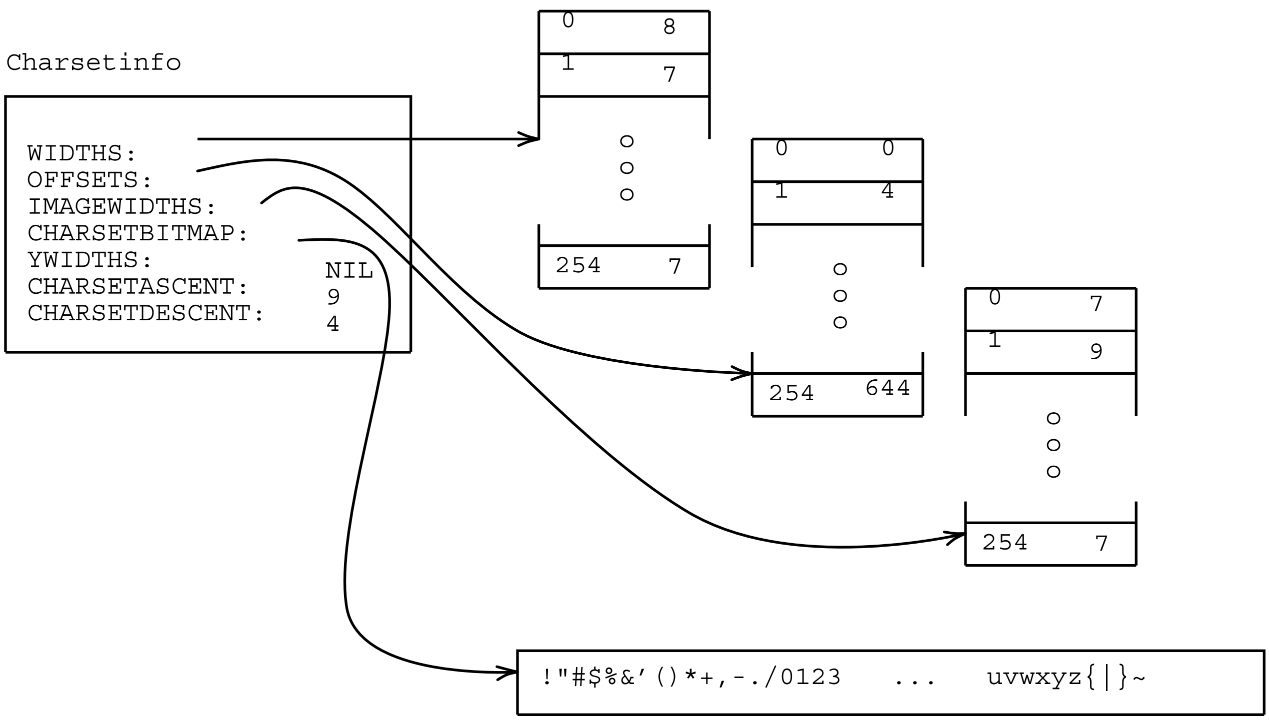
Internals

Note: The information in this section is advisory only. There is no guarantee of back-compatibility in future changes.

Structure of Font Descriptors

To incorporate the increase in information contained in font descriptors due to NS characters, the structure of font descriptors has changed. The current structure is as follows.


Font Descriptor
FONTFAMILY: Modern
FONTSIZE: 12
FONTFACE: (Medium Regular Regular)
ROTATION: 0
FONTDEVICE: RandomDIGdevice
\SFAscent: 10
\SFDescent: 4
\SFHeight: 14
FONTCHARSETVECTOR:
CharsetVector
0
1
254



Description:

Font Descriptors:

The figure represents the logical structure of a font descriptor. Clarification of field values:

Face: a list containing the weight, slope and expansion. (expansion is always regular)

Rotation: degrees of rotation

Device: name of the device for which this font is created

Ascent: the maximum distance above the baseline for any character in this font.

Descent: the maximum distance below the baseline for any character in this font. (always positive)

Height: maximum total height of any character in this font. Equals Descent+Ascent.

FontDeviceSpec: this is the font specification actually used to create this font after coercions. Thus, if the fontcreate method substituted something other than the original arguments to fontcreate, then this field shows the real contents of the font descriptor, while the family (etc.) fields contain the ostensible (pre-coercion) contents. For instance, display font substitions occur in \CREATECHARSET.DISPLAY according to the variable MISSINGDISPLAYFONTCOERCIONS.

OtherDeviceFontProps: available to the implementation of each stream. Unexamined by the system.

Charsetvector: this is a ptr block pointing to the individual charsetinfo's for each of 255 charsets. These are either NIL or a ptr to the charsetinfo. In the diagram, only the charsetinfo for charset 0 is present. When a fontdescriptor is created the charsetvector will already be present.

Charsetinfo:

A charsetinfo contains all the metrics for a single character set (255 characters) of the font.

Widths: a block of SMALLP's which gives the width in device units for each character (how much to advance the stream xpos after printing this character). This field must be present no matter what the stream type.

Offsets: integer offsets into the bitmap field, showing the xposition of the beginning of the bitmap for this character. Many streams (especially hardcopy streams) have no need of a bitmap or offsets.

Imagewidths: This is the width of the image of this character. Often this will be the same as the width, but it can be less or greater. The field must always be valid.

Bitmap: The bitmap for the characters in this charset. [optional]

YWidths: currently unused, but for forward compatibility, make this be a block giving the y distance to move for each character (i.e. carriage return should move the y position the height of the font.) [**?** is this a positive distance]

Ascent: max ascent in the character set.

Descent: max descent in the character set.

Widths, Offsets, YWidths and Imagewidths are instances of the result of CREATECSINFOELEMENT (see below).

Exported Macros and Functions

(\CHARSET *CHARCODE*)
returns the character set (upper 8 bits) of *CHARCODE*.

(\CHAR8CODE *CHARCODE*)
returns the offset of *CHARCODE* within the character set (the low 8 bits of the character)

(\CREATECHARSET *CHARSET FONT NOSLUG?*)
the createcharset method (determined by the value of the variable IMAGESTREAMTYPES) is called. *NOSLUG?* determines the result if the createcharset method returns NIL. If *NOSLUG?* is NIL, then a "slug" charsetinfo (all characters have a slug (black rectangle) as their image) is returned, otherwise NIL is returned. [This needs improvement, to control how a slug is build. Currently \BUILDSLUGCSINFO is presumed to know how to build a slug for all imagestreams.]

(\GETCHARSETINFO *CHARSET FONTDESC NOSLUG*?)
returns the charsetinfo for *CHARSET* (0..254) from *FONTDESC*. Calls \CREATECHARSET if the charsetinfo wasn't cached already.

(\SETCHARSETINFO *CHARSETVECTOR CHARSET CSINFO*)
will install *CSINFO* as the charsetinfo of (smallp) *CHARSET* in *CHARSETVECTOR*. Since \CREATECHARSET calls \SETCHARSETINFO directly, it usually need not be called.

(\CREATECSINFOELEMENT)
creates a word block for installing as a widths (imagewidths, offsets) field in a csinfo.

(\FGETWIDTH *WIDTHSBLOCK CHAR8CODE*)
returns the smallp width at index *CHAR8CODE*. e.g. (\FGETWIDTH (FETCH (CHARSETINFO WIDTHS) OF csinfo) 55)

(\FSETWIDTH *WIDTHSBLOCK CHAR8CODE WIDTH*)
sets the smallp width at index *CHAR8CODE*.

(\FGETCHARWIDTH *FONTDESC CHARCODE*)
returns the width of any character without having to explicitly fetch the correct charsetinfo for the character set of the character.

(\FGETIMAGEWIDTH *FONT CHARCODE*)
analagous to \FGETWIDTH but for imagewidths (the width of the character image rather than the amount the xposition should be incremented when printing this character.)

(\FGETCHARIMAGEWIDTH *FONT CHARCODE*)
analagous to \FGETCHARWIDTH but for imagewidths.

(\FGETOFFSET *OFFSETBLOCK CHAR8CODE*)
analagous to \FGETWIDTH but for offsets (the position in the bitmap for this character set where the image for this character begins.)

(\FSETOFFSET *OFFSETSBLOCK CHAR8CODE OFFSET*)
sets the smallp offset at index *CHAR8CODE*.

1109 vs 1188 Floating Point Benchmark Results

Jan Pedersen
28 Aug. 1986

The timing results below compare the performance of an 1109 vs an 1188 on a suite of floating point benchmarks. The desire was to measure as closely as possible, using TIMEALL, the relative speeds of various arithmetic opcodes. No attempt was made to benchmark a "real" (e.g. linear algebra) application.

The 1109 was running a lispcore sysout(Makesysdate "21-Aug-86"), a real memory size of 7167 pages, and a set of Weitek floating point chips.

The 1188 was running a lispcore sysout(Makesysdate "25-Aug-86"), a real memory size of 7424 pages, and no floating point hardware, but microcode support for serveral boxed and unboxed floating point opcodes.

Both boxed and unboxed opcodes were benchmarked. Most benchmarks were a tight loop with the opcode evaluated 10,000 times. The block floating point opcodes were evaluated 1,000 times on arrays of size 100 (for a total of 100,000 arithmetic operations). Some of the boxed opcodes produced no garbage since they returned one of their inputs as an output, or returned T or NIL.

The 1188 had no microcode support for the block opcodes (they ran in lisp using scalar unboxed opcodes).

Cpu time and GC time are recorded separately for the boxed opcodes. Cpu time and CPU less the CPU time for an empty loop are recorded separately for the unboxed opcodes.

NA stands for Not Applicable.

Boxed Float Results (Time in seconds)

	1109		1188		Ratio	
Opcode	Cpu	Gc	Cpu	Gc	Cpu	Gc
FPLUS	.98	(2.35)	1.02	(2.05)	.96	(1.15)
FDIFF	.98	(2.35)	1.03	(2.05)	.96	(1.15)
FTIMES	.99	(2.35)	1.17	(2.05)	.85	(1.15)
FQUOT	1.36	(2.34)	1.19	(2.05)	1.14	(1.15)
FGREATP	.304	(0.0)	.267	(0.0)	1.14	(NA)

	1109		1188		Ratio	
Function	Cpu	Gc	Cpu	Gc	Cpu	Gc
FABS	2.1	(2.33)	2.14	(2.03)	.98	(1.15)
FMINUS	1.13	(2.33)	1.11	(2.04)	1.02	(1.14)
FIX	6.56	(0.0)	5.85	(0.0)	1.12	(NA)
FMAX	1.15	(0.0)	1.04	(0.0)	1.10	(NA)
FMIN	1.14	(0.0)	1.02	(0.0)	1.12	(NA)

Unboxed Float Results (Time in seconds)

	1109		1188		Ratio	
Opcode	Cpu	(- empty)	Cpu	(- empty)	Cpu	(- empty)
Empty lp	.109	(NA)	.097	(NA)	1.12	(NA)
UFPLUS	.244	(.135)	.363	(.266)	.67	(.508)
UFDIFF	.244	(.135)	.362	(.265)	.67	(.509)

UFTIMES	.26	(.151)	.515	(.418)	.50	(.361)
UFQUOT	.616	(.507)	.533	(.436)	1.16	(1.16)
UFGREATP	.235	(.126)	.206	(.109)	1.14	(1.16)
UFABS	.178	(.069)	.161	(.064)	1.11	(1.08)
UFMINUS	.179	(.07)	.161	(.064)	1.11	(1.09)
UFIX	.213	(.104)	.205	(.108)	1.04	(.963)
UFMAX	.235	(.126)	.206	(.109)	1.14	(1.16)
UFMIN	.231	(.122)	.206	(.109)	1.12	(1.12)
BLKPLUS	.39	(.281)	6.73	(6.63)	.058	(.042)
BLKDIFF	.384	(.275)	5.71	(5.61)	.067	(.049)
BLKTIMES	.39	(.281)	7.63	(7.53)	.051	(.037)
POLY	.45	(.341)	4.92	(4.82)	.091	(.071)

Summary
----

a.) Unboxed operations are a factor of ten faster than boxed operations across the board.

b.) On an 1109 the block opcodes yield another factor of five to ten.

c.) For scalar operations, the 1188 is never worse than .36% of the 1109, and never better than 1.16% of the 1109.

d.) The 1188 was actually faster than the 1109 for several unboxed opcodes -- and generally faster for the boxed opcodes.

e.) The 1109's floating point hardware really comes to the fore in the block opcodes. Unfortunately, with the exception of polynomial opcode, these opcodes are rarely used.

J.P.

Freemenu Internal Documentation

All names are in the Freemenu package (except old names, which begin with \\fm.)

The Freemenu Description Language:

A freemenu description is defined by:

[]   optional
{}   group of things for some other operator
*   0 or more
+   1 or more
| or
       literals in bold

menu-desc --> menu-element ; can be passed to **freemenu**

menu-element --> item-desc **|** group-desc

item-desc --> **(**{item-prop value}* **:label** item-label {item-prop value}***)**

item-prop --> one of the prop keywords described in the user doc

item-label --> string **|** bitmap **|** imageobj

group-desc --> **(**[**:group**] [prop-spec] {menu-element}***)**

prop-spec --> **(:prop** {group-prop value}***)**

group-prop --> one of the group property keywords described in the user doc

Group Formats:

The possible formats are:

:column - the elements in this group are layed out vertically, top to bottom, flush left

:row - the elements in this group are layed out horizontally, left to right, along the same baseline

:table - the elements in this group are layed out vertically, top to bottom, flush left. additionally, the second element of each direct subgroup is positioned at the same horizontal location. the third element of each direct subgroup is positioned at the same horizontal location. and so on, such that each element of each subgroup is both in a row and a column.

:explicit - each element of this group has a :left and :bottom property specifying its position. if the group property :coordinates is :group, then the values of the :left and :bottom property are relative to the lower left corner of the group, and if :menu (the default) they are relative to the lower left corner of the topmost group.

Group Format Defaults:

The format of a group of menu-elements can be specified in the prop-spec for that group. If it is not, the following rules apply:

- The default format for a group-desc passed directly to **freemenu** is :column.

- When formatting a group in :column format, the default format of each sub group is :row.

- When formatting a group in :row format, the default format of each sub group is :column.

- When formatting a group in :explicit format, the default format of each sub group is :explicit

- When formatting a group in :table format, the default format of each sub group is :table-element, which signals the formatter to horizontally align each element in the sub group with the other elements in the table. If the format is specified for a sub group, the elements of that sub group will not be aligned with the other elements in the table.

The Formatter:

Entry point:

The entry point to the freemenu formatter is \\fm.format (fm::format). It takes a description of the menu to be formatted and returns a group hierarchy structure. In the current version of freemenu, the group structure is just an alist of group id's and properties, with the topmost group first. The formatter takes the following arguments:

description : a menu-element as defined above

format : the format to be used to lay out this group

font : the default font for each item in this group, must be a fontdescriptor

left, bottom : the lower left corner of the group. format everything relative to this position.

rowspace : the number of pixels to leave between rows, that is, the space to leave between elements in a column

columnspace : the number of pixels to leave between columns, that is, the space to leave between elements in a row

mother : the mother group of the one being passed for formatting. in the current version, this is the ID of the mother group, not the group itself.

The remaining arguments are optional. In the current version they are not specified, but they are SET by the guy who processes the group prop-spec. They might want to be specified in later versions:

id - the ID to use for this group

props - the prop-spec to use for this group

Group prop-spec:

The macro \\fm.setupprops processes the group prop-spec in the description and fills in the slots in group accordingly (currently plist format). At the same time, it sets the arguments above from the values specified in the prop-spec, thus overriding the passed-in default. When the formatter is done with the current group, the function return, and thus all of the arguments are popped off the stack, and the previous state is now dynamically visible. This is how the formatter keeps track of format prop state, and pops back to the previous state when done formatting a group.

\\fm.setupprops takes a group-spec and a list of group props to set. It generates code that will fill in the group props information, and then sets the format state arguments for the props that are in the list of group props to set.

The :left and :bottom group props in the prop-spec specify offsets for the entire group from where the formatter would otherwise position the group. This is similar to the way :left and :bottom item props specify offsets for an item that is automatically formatted (as opposed to explicitely positioned in the menu description).

Group boxing:

The macro \\fm.checkforbox looks in the props to see if the group is boxed, and if it is, it adjusts the left,bottom position of the group to allow for the width of the box and the boxspace (boxoffset = box-width + box-space).

Then the elements in the group are layed out normally.

Finally, the macro \\fm.updateforbox, if the group is boxed, does the following: save the calculated extent of the group as the interior region of the group, and then adjusts the region to include the box, and saves this region as the region property of the group.

Layout routines:

The formatter calls one of the layout routines (\\fm.layout-column \\fm.layout-row \\fm.layout-table \\fm.layout-explicit) to lay out the elements in the group. The layout routines provide the real guts of the formatting. These functions return multiple values:
1. list of items in all the elements layed out
2. list of groups in all the elements layed out
3. extent region of all the elements layed out
4. list of id’s of all the subgroups layed out (this one would go away if had real group structure, instead of flat alist)

Layout algorithm:

Here is a description of the algorithm used in laying out a column of elements. The other routines follow the same procedures, but operate in a different dimension or have other overhead (like layout-table).

```
bind extent ; the region the elements in this group occupy
        itemlist ; a flattened list of items in this group
        grouplist ; a flattened list of groups in this group
        subgroupids ; a list of id's of groups in this group
        element-position ; the position of the next element formatted
        element ; temp for hanging onto the element newly formatted

iterate through each element in the group description
  if the element is a group description:
        set element to call format on the group description
        extend: extent, itemlist, grouplist, subgroupids with the results
  otherwise:
        set element to create an item from the item description
        extend extent, itemlist with the item

  increment element-position by the size of element and extra space

return extent, itemlist, grouplist, subgroupids
```

Putting it all together:

With the information returned by the layout routine in the hands of the formatter, create a new group structure from the group prop-spec and extent, itemlist, and subgroupids. Add this group to the front of the list of groups layed out, and you get a list of groups for the description just formatted.

Freemenu Data Structures:

A Freemenu is currently a window, with all of the necessary properties set to make it behave as a menu when it is open. Eventually a Freemenu probably wants to be an independent structure, which can be enclosed in different display mechanisms, like windows, image-objects, pop up managers, etc.

There are three main data structures composing a Freemenu:

ITEM - Instance of the datatype freemenuitem, one for each item in the menu. The macro itemprop provides access to fetching and replacing fields in the datatype. The macro %itemprop is an internal version of the same macro which doesn't type check the item and requires the field (property) name to be provided explicitely (not bound to some variable) in the call.

GROUP - A list structure describing a group of items in the menu. List fomat is (<group-id> <group-type-identifier> {<prop> <value>}*). The <group-id> is used for ASSOC purposes on the list of all groups. The <group-type-identifier> is checked by the macro group-p to ensure that this list is a valid freemenu group. The macro groupprop provides access to getting and setting the props in the cddr of the list.

NWAY - A list structure describing an nway collection in the menu. The list format is the same as that for groups. nway-p and nwayprop are analogous to group-p and groupprop.

A list of all the items in the menu is stored on the ITEMS window property.

A list of all the groups in the menu is stored on the GROUPS window property.

A list of all the nway collections in the menu is stored on the NWAYS window property.

The functions get-item, get-group, and get-nway take an id and search the appropriate list for a matching freemenu item, group, or nway, respectively.

Additionally, many of the freemenu functions, like redisplay-menu, depend on being able to use a flat list of objects in the menu, either items, groups, or nway collections.

GC Overview:
February 6, 1986
Jan Pedersen & Greg Nuyens

Reference counts are not contiguous with their objects, but rather are kept in three tables: \HTMAIN (the hashed main ref count table), \HTCOLL (the collision table), and \HTBIGCOUNT (the big ref count table). \HTMAIN is a hash table, where the hash function is based on the address of the object, while \HTCOLL and \HTBIGCOUNT handle two aspects of hash table overflow.

Since most (?) objects have refcount 1 (e.g. cons cells of a list), they are not explicitly represented; if an object (of a refcountable type) is not present in the tables, its refcount is 1.

If several objects hash to the same entry in \HTMAIN, then the entries are kept in a linked list of ref count entries in \HTCOLL.

If an object has a refcount equal to \MAXHTCNT (63), its refcount is stored in a "big" refcount table (\HTBIGCOUNT).

The Lisp function \HTFIND can handle all cases and is the punt function for the opcode (GCREF), which handles only the simplest case of no collision and no big refcount. Many opcodes call GCREF as a microcode subroutine and if GCREF punts, an entry is made in the table \HTOVERFLOW. Before opcode completion, the microcode calls the Lisp function \GC.HANDLEOVERFLOW, which processes \HTOVERFLOW by explicitly calling \HTFIND on each entry.

\HTMAIN is a locked down table of 32K word sized entries (or 64K bytes).

\HTCOLL is a paged table of 32K double word sized entries (or 128K bytes), where the first word in a pair is a ref count entry and the second is a 16 bit offset to the next entry in the chain.

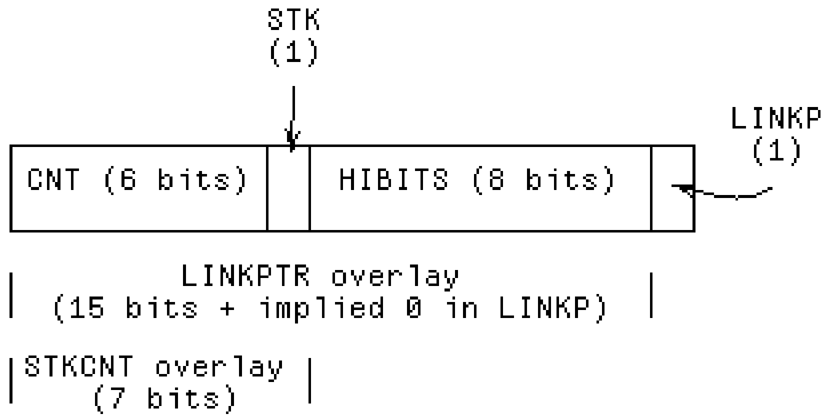
\HTBIGCOUNT is a linearly searched table of big ref count entries; new pages are allocated as needed.

Hashing:

Given an address, the hash function computes a 15 bit offset into \HTMAIN by logically shifting the low 16 bits of the address right one bit.

GC record structure:

Each entry in \HTMAIN is a word (16 bits) long, and is a record of type GC. Its structure is the following:



REFCOUNT TABLE ENTRY

CNT is the refcount (0 to 63, excluding 1)
STK is on if this object is referenced from the stack.
HIBITS contains the top 8 bits of the PTR address of the object (the offset at which you found this entry gives you the lower 15 bits (only even objects are refcounted)).
LINKP is on if this entry is a pointer to the collision table.

Organization of \HTCOLL:

\HTCOLL points to a double word entry, the first word of which is a 16 bit offset from \HTCOLL where the first entry on the linked free list may be found. The secord word is a 16 bit offset from \HTCOLL where the first free double word in sequential storage may be found.

At startup the first word is zero, indicating an empty free list and the second word is four, indicating that the first free double word in sequential storage is at offset four from the top of \HTCOLL.

A new entry is allocated by first looking at the free list. If the free list is empty, a new entry is allocated from sequential storage.

It is an error to allocate double words from sequential storage at offsets greater than or equal to \HTCOLLTHRESHOLD (65528 or 177770Q) -- if this occurs the garbage collector is turned off.

The double word entries in \HTCOLL are similar to those in \HTMAIN in that the first word in a pair is an instance of record type GC. The adjacent word is a 16 bit link offset (field name NXTPTR) from the top of \HTCOLL to the next double word entry in this collision chain. A NXTPTR value of zero indicates end-of-chain.

Algorithm:

\HTFIND is called with two arguments, PTR (a pointer to the object) and CASE (oneof \ADDREFCASE, \DELREFCASE, \SCANREFCASE or \UNSCANREFCASE). The \SCANREFCASE means mark the object as referenced on the stack, likewise \UNSCANREFCASE. \HTFIND returns PTR if the result is zero ref count, else NIL.

First, \HTFIND hashes the address into a 15 bit offset from the top of \HTMAIN, pointing at a refcount table entry. If the resulting entry is empty (all bits 0) then the macro .NEWENTRY. handles it.

.NEWENTRY. does the following:
updates the HIBITS.
and by case does:

- \ADDREFCASE sets CNT to 2 (since its absence from the table implied previous refcount 1)
- \DELREFCASE leaves the CNT at 0 and returns the PTR
- \SCANREFCASE Sets count to 1 and sets the STK bit.

If the entry from the hash has the same HIBITS then this main table entry is the entry for PTR, so the macro .MODENTRY. handles it.

.MODENTRY. does the following:
if CNT is \MAXHTCNT, then this entry is superceded by an entry in the big refcount table (\HTBIGCOUNT), so call \GC.MODIFY.BIGREFCNT.
Otherwise, by refcount case:

- \ADDREFCASE if incrementing CNT makes it equal to \MAXHTCNT, then call \GC.ENTER.BIGREFCNT, otherwise just increment CNT.
- \DELREFCASE decrements CNT
- \SCANREFCASE sets STK.
- \UNSCANREFCASE clears STK.

If resulting refcount is 1and no STK (STKCNT = 2), return T signalling removal.

If the HIBITS don't match, then a new collison has occured. The collision resolution is basically to construct a linked list of refcount table entries out of cells from \HTCOLL

New collision:
-- Get two double word entries (LINK and PREV) from \HTCOLL by calling the .GETLINK. macro.
-- Make the ref count contents of PREV equivalent to the ref count contents of the entry in \HTMAIN and link PREV to the next double word entry, LINK.
-- Smash the \HTMAIN entry so that the LINKPTR overlay is an offset to PREV.
-- Mark LINK as empty, and the end of a chain.
-- proceed with LINK as an empty entry (.NEWENTRY.)

> where .GETLINK. does the following
> -- fetch the FREEPTR field of \HTCOLL (should be an offset to the front of the free list).
> -- if FREEPTR is non-zero, fetch the first free cell and update the FREEPTR field of \HTCOLL
> -- else fetch the NEXTFREE of \HTCOLL (should be an offset to the first free double word in sequential storage). If NEXTFREE is not GEQ to \HTCOLLTHRESHOLD, allocate that double word and increment NEXTFREE by 2.

If the entry from the hash has LINKP set, then a collision has already occurred and the entry is a pointer to a chain in the collision table. The LINKPTR overlay is used as a (16 bit) offset into \HTCOLL (the first 15 bits of the entry are used as is, with the last bit (LINKP) masked to zero). Note that the LINKP test should occur before testing HIBITS, but we place the explanation here since this case is more complex than a new collision.

So LINKPTR is an offset to the beginning of a chain of double word entries in \HTCOLL. This chain is searched sequentially (by following NXTPTR offsets down the chain), resulting in either finding an entry for PTR, or hitting the end of the chain (NXTPTR offset equal to zero).

If an entry is found, then .MODENTRY. handles it as before except that if the refcount goes to zero the link is deleted from the chain (by .DELLINK.).

> .DELLINK. takes three args, (LINK -- the link to be deleted, PREV the previous link in the chain (can be NIL), and ENTRY -- the \HTMAIN table entry)
> -- If PREV, than update PREV to point to the link following LINK, else update ENTRY to point to the link following LINK.
> -- Place LINK back on the free list (.FREELINK.)
> -- If ENTRY now points to a chain one entry long, then update ENTRY to have the refcount table entry contents of the remaining link, and free the remaining link.
>
> > .FREELINK. takes one arg, the LINK cell to be freed
> > -- zero LINK
> > -- place LINK on free list of \HTCOLL by update the first word (field name FREEPTR) of \HTCOLL

If no entry in the chain is found for PTR, then allocate an entry (.GETLINK.) and put it on the end of the chain. Treat the new LINK as an empty entry (.NEWENTRY.).

Stay tuned for further updates...

Harmony Release Message to Internal Xerox Users

(this text saved: {eris}<lisp>Harmony>Doc>HarmonyReleaseMsg.txt)

This message announces the latest release of Interlisp-D to the Internal Xerox community. This release is known as the "Harmony" release.

The Harmony release is a significant improvement over the last Interlisp-D release (Carol), with changes and improvements in a great number of areas. Over 450 ARs have been closed between Carol and Harmony.

Full release documention of differences between Carol and Harmony is still in preparation. A draft version is stored in {eris}<Lisp>Harmony>Doc>HarmonyReleaseSpecifics.txt.)

How to Install Harmony

For information on installing the Harmony release of Interlisp-D on your machine, see {eris}<Lisp>Harmony>Doc>GettingStarted.tedit (& .press).

VERY IMPORTANT WARNING FOR DLION USERS: The Harmony release of Interlisp-D is compatible with the Mesa 11.0 (Klamath) version of the Pilot DLion file system. This release of Pilot file system is INCOMPATIBLE with older versions. This means that if you use the Mesa 11.0 tools on a Dlion with an old file system, it is possible to DESTROY information on your Dlion disk. For example, if you start Othello 11.0 on a machine configured in Mesa 10.0, it may print a message saying that "the disk needs to be scavenged " -- if you scavenge the disk, you may lose the information on it. Before upgrading to Mesa 11.0, be sure that all valuable information on your Dlion disk is stored on a file server, or on floppy disks.

Harmony Release Files

All of the files needed to use the Harmony release are in the directory {eris}<Lisp>Harmony>. Specifically:

{eris}<Lisp>Harmony>Basics>*
Contains the Harmony sysouts, initial microcode, sample INIT files (INIT.SAMPLE, INIT.NONET, etc.).

{eris}<Lisp>Harmony>Library>*
Contains all of the supported lisp library packages. Most of these have been revised since the Carol release

{eris}<Lisp>Harmony>Lispusers>*
Contains only a few files. These files are new versions of Lispusers packages which have been updated to run in Harmony. Most of the files on {eris}<lispusers> can run in Harmony unchanged.

{eris}<Lisp>Harmony>Mesa>*
Contains some tools written in Mesa which are useful when running Interlisp on a Dlion (1108). These include tools for partitioning a Dlion disk, and installing Interlisp. These tools are described in {eris}<lisp>Harmony>Doc>GettingStarted.tedit.

{eris}<Lisp>Harmony>CM>*
Contains the command files used to partition Dlion disks, and install Interlisp onn a Dlion, from Othello or the new "Hello" tool.

{eris}<Lisp>Harmony>Doc>*
Contains documentation files.

Important Changes in Harmony
============================

Please see the full release documentation {eris}<Lisp>Harmony>Doc>HarmonyReleaseSpecifics.txt. There are two important changes which are summarized here.

Printing and Fonts:
The printing and font mechanisms have been extensively revised and cleaned up. Everything that worked before should still work, plus a lot of new functionality has been added. IMPORTANT CHANGE: The global variables which point at font files and font widths files have been changed. Instead of FONTDIRECTORIES, STARFONTDIRECTORIES, etc., the appropriate variables are DISPLAYFONTDIRECTORIES, DISPLAYFONTEXTENSIONS, PRESSFONTWIDTHSFILES, and INTERPRESSFONTDIRECTORIES. If these variables are not set correctly, you may get FONT NOT FOUND errors when you try to print or display characters in different fonts. The file {eris}<lisp>Harmony>Basics>INIT.CIS is a site-specific init file which contains values for all of the variables needed. This file should be modified for each individual site.

Dlion Local File system:

As mentioned above, the Harmony release of Interlisp-D is compatible with the Klamath version of the pilot file system. In order to use the Interlisp-D on a Dlion, it is necessary to upgrade your Dlion to Klamath. While Mesa provides the ability to convert Mesa 10.0 workstations to Mesa 11.0, this "forward scavenge" will NOT convert Interlisp-D disk partitions. The simplest and most fail-safe way of converting your disk is to copy any valuable files from the local file system to a file server or floppies (using Carol Interlisp), upgrade to Klamath, and reload the files.

Besides the changes in the local file system format, a few of the functions have been renamed (MKDIR -> DFSCREATEDIRECTORY, MAKEPILOT -> DFSPURGEDIRECTORY). Also, the mechanism for specifying a particular volume on a Dlion has changed: instead of using {FOO}BAR to access the file BAR on the local disk volume FOO, one should use {DSK}<FOO>BAR. For compatibility, {DSK}FOO will search and use the first local file system volume on the disk after the currently-running Interlisp volume. In the Othello command files provided for partitioning the Dlion disk, the default name of the local file system volume has been changed from DSK to LispFiles.

February, 1989

Movement of Guaranteed Type Numbers, Addition of hashing MISCN subops.

Summary

The type numbers of several Lisp datatypes were moved down into the range that is "known to the microcode," to allow me to write C support for hashing.

Overview

Certain lisp type numbers must be known to the underlying implementation (microcode on 1186's, the C emulator on Suns). One obvious example is that the emulator must be able to detect SMALLPs, so it can do arithmetic quickly.

There is a range of type numbers that are allocated very early in the load-up process, so that they are assigned known numbers. After those "well-known" types, the type numbers for hunked storage are allocated. After that, type numbers are allocated to types as the loadup progresses, in whatever order they are defined.

The New Requirement

I implemented MISCN sub-opcodes for CL:SXHASH, CL::EQLHASHBITSFN, IL:STRINGHASHBITS, and IL:STRING-EQUAL-HASHBITS. CL:SXHASH has special-case code for several data types that were not in the "well-known" range: RATIONAL, COMPLEX, PATHNAME, and BIGNUM. I needed to move the type numbers for those types down.

The New "Well-Known" Type Numbers

The well-known type numbers are defined from the list \BUILD-IN-SYSTEM-TYPES, which is defined in the file LLDATATYPE. Listed below are the old and new type number assignments:

Pre-existing		**Newly-Added**	
1	SMALLP		
2	FIXP		
3	FLOATP		
4	LITATOM		
5	LISTP		
6	ARRAYP		
7	STRINGP		
8	STACKP		
9	CHARACTER		
10	VMEMPAGEP		
11	STREAM		
12	BITMAP		
13	COMPILED-CLOSURE		
14	ONED-ARRAY		
15	TWOD-ARRAY		
16	GENERAL-ARRAY		
		17	BIGNUM
		18	RATIO
		19	COMPLEX
		20	PATHNAME

Changes to the C emulator

Moving those four types into the well-known range had the effect of moving the array hunk type numbers up. The C coded garbage collector had several hunk type numbers hard-coded into it (as hex constants!!). I added definitions for all the type numbers in use (both well-known and hunk) to LISPTYPES.H, and changed the GC code (in GCRECLAIMCELL.C, as I recall) to accomodate it.

The new hashing MISCN sub-opcodes

The source for the new opcodes is in sxhash.c. The document {Eris}<Lispcore>Internal>Doc>Opcodes.TEdit has been updated to reflect their addition. I have also reserved opcodes for CL:VALUES and CL:VALUES-LIST.

The Lisp definitions for the functions CL:SXHASH, CL::EQLHASHBITSFN, IL:STRINGHASHBITS, AND IL:STRING-EQUAL-HASHBITS are now written to use the MISCN sub-opcode appropriate; the UFNs are defined in the same files as the original functions, CMLHASH and LLARRAYELT.

Hello 11.0

```
program:        [eris]<lisp>Harmony>mesa>HelloDlion.boot
documentation:  [eris]<lisp>Harmony>Doc>Hello.tedit
```

Hello 11.0 is a modified version of the Othello 11.0 utility, which is used for managing Pilot disk volumes. It offers most of the Othello commands (documented in the Mesa Users Guide), and adds a few commands which are useful when using Interlisp-D on a Dlion.

Hello is a .boot file, which can be loaded onto a Dlion disk using the Fetch Boot File command of Othello. It can also be booted off of a floppy disk.

Loading Interlisp From a File Server to a Logical Volume:

When Hello starts up, it prints out some information about the machine it is running on, including the Dlion's host number and memory size, and then prints the prompt ">", to indicate it is ready to recieve a command. The "Online" command, which is automatically printed, tells Hello to bring the physical disk on-line. (Note: all user input is underlined, including confirming carriage-returns)

```
Hello 11.0 of 6-Sep-84 10:14:03
Processor = ...
Memory Size=1536 bytes
>Online
Drive Name: RD0
```

Before fetching a lisp sysout from a file server, it is necessary to open a connection to a file server, and login. Note: Hello currently cannot communicate with NS file servers (ones with colons in their names, such as "Phylex:").

```
>Open
Open Connection to ERIS
>Login
User: Sannella
Password: *****
```

In order for a lisp sysout to run, it needs to have a special "initial microcode" file installed. This microcode only needs to be installed once for each Dlion.

```
>Initial Microcode Fetch
Drive Name: RD0
File Name: [eris]<lisp>Harmony>Basics>Lisp11SAx000Initial.db
Formatting...Fetching...Installing...done
```

Now, fetch the Interlisp sysout file [eris]<lispcore>next>Full.sysout and store it on the logical volume named "Lisp". Depending on the size of the sysout file, and the load on the ethernet, this can take 5-10 minutes.

```
>Lisp Sysout Fetch
Logical volume name: Lisp
Lisp sysout file name: [eris]<lisp>Harmony>Basics>Full.sysout
Fetching....
```

Before running a lisp sysout, it is necessary to "expand" the file containing the sysout to the full size of the logical volume. This will allow Interlisp virtual memory to grow as Interlisp needs more space. If this is NOT done, there can be problems with Interlisp on large-memory Dlions. Eventually, the low-level Interlisp virtual memory management system will be improved, so this will not be necessary.

```
>Expand Vmem file
volume to expand: Lisp
```

Finally, the Interlisp image on volume "Lisp" can be started with the boot command.

```
>Boot
```

```
Logical volume name: Lisp
Boot Lisp from this volume? YES
```

Copying Interlisp From One Logical Volume to Another:

A very useful facility that Hello offers is the capability of copying an Interlisp sysout from one logical volume to another. This is much faster than retrieving a sysout over the ethernet. Many people like to keep a "virgin" sysout on their Dlion, and reload from that. Here is how it is done: Load a sysout onto a volume named "BootLisp", as specified above. This sysout will be the "virgin" sysout used to re-initialize other Interlisp volumes. This volume should not be expanded with the "Expand Vmem file" command, and should never be booted.

```
>Open
Open Connection to ERIS
>Login
User: Sannella
Password: *****
>Initial Microcode Fetch
Drive Name: RD0
File Name: [eris]<lisp>Harmony>Basics>Lisp11SAx000Initial.db
Formatting...Fetching...Installing...done
>Lisp Sysout Fetch
Logical volume name: BootLisp
Lisp sysout file name: [eris]<lisp>Harmony>Basics>Full.sysout
Fetching....
```

To re-initialize the Interlisp volume "Lisp", copy the sysout file from the volume "BootLisp". This takes about a minute.

```
>Copy Lisp from Another Volume
Volume to copy from: BootLisp
Volume to copy to: Lisp
```

Now, expand and boot the "Lisp" volume as if the sysout had been loaded from a file server.

```
>Expand Vmem file
volume to expand: Lisp
>Boot
Logical volume name: Lisp
Boot Lisp from this volume? YES
```

Determining What is on Your Disk:

It is easy to forget exactly what sysouts are on what volumes of the DLion disk, or which Interlisp sysouts have been "expanded", etc. The "Describe" command prints out useful information about every volume on the disk.

```
>Describe
   .
   .
Volume Lisp (type=normal) 700 of 16200 pages free
  starting at physical address 4129
  Lisp sysout: [eris]<lispcore>next>Full.sysout (3-Aug-84 10:35:17)
   .
   .
```

In this example, we can tell what lisp sysout, with which creation date, is stored on the volume "Lisp". The fact that 700 of 16200 pages are free strongly indicates that the sysout has been expanded (expansion doesn't use ALL of the space).

Additional information on "How-To" can be found on:

{Eris}<Sybalsky>How-to>

such as:

Installing New 1108 ucode
Running AR cleanup
Seding info to Sales
Writing AR test cases

Subject: How to write the release notes
To: James
cc: Sannella.pa

How to write a release note for a fixed AR:

When an AR is fixed, it is important to write a release note for the AR, even if it is just a note saying "Release Note: none needed". There is no better time to do this than when the problem is still fresh in your mind. And there is no better person to do this than the implementer, who knows exactly how important the AR fix is.

Does this AR affect the next release? There are some ARs which are created and closed during the course of development. However, it is hard to tell just from the text of the AR. It is important to put this information in the AR, so the Release master can tell that it shouldn't be collected. The note "Release Note: Don't release -- development bug" is better than nothing.

Does this AR affect outside users? If an AR just affects internal users, put a release note "Release Note: Don't release -- internal info" in the AR.

If there is info in the AR that the outside users should know, compose a release note. It doesn't have to be perfect, but the implementer is in a better position to know what needs to be told than anyone else.

A single release note item should be talk about one piece of information. If a single AR contains or refers to multiple subjects, compose multiple release notes.

A release note contains a title, telling the substance of the item. Like AR subjects, the release note title should be short and as informative as possible (think of newspaper headlines). Include as many keywords as possible. For example "* New window functions: FOOBAR, BAZ" is better than "* New window functionality". When documenting bug fixes, try to make the titles "positive", saying "* FOOBAR arguments interpreted correctly" rather than "* Bug fixed where FOOBAR ignored argument".

When in doubt about whether a bug fix or new feature is worth documenting in the release notes, please document it. It is a lot easier for the release master to throw away information than to write it from scratch.

How to assemble individual items into the "Release Notes":

If the individual release note items have been written, the primary job is to sort them by subject and importance, and assemble the big document.

For the Harmony release, the "topics" were ordered loosely on the importance of the different areas for that release (a lot of changes were made to I/O, fonts, printing, so they came first). This order is not sacred.

Within each topic area, the items are sorted by importance to the user, a subjective decision. Major new functionality, and incompatible changes obviously want to come first.

Characteristics of the Xerox 1100 Machines upon which the Gabriel Benchmarks Were Performed

All three members of the Xerox 1100 family are custom microcoded processors. The Interlisp-D virtual machine is built around a compact 8-bit "bytecode" instruction set, the opcodes of which are implemented by a combination of microcode and macrocode. Not all bytecodes are supported directly in each member by microcode; the alternative is a trap out to a standard Lisp function. Above the level of the instruction set, all three members of the family appear identical to the Interlisp-D programmer. The implementation is such that a memory image can be compatibly run on any of the machines, without any change.

An Interlisp pointer is an address in a 24-bit virtual adress space; a "quantum map" indexed by the high bits of the address provides information for type decoding. Additionally, litatoms (symbols) and immediate numbers (integers in the range of -2^16 to 2^16-1) live in a reserved portion of the address space; integers of larger magnitude (within the range -2^31 to 2^31-1) are "boxed"; floating-point numbers, which are in IEEE 32-bit format, are also boxed. All three machines have a 16-bit memory bus and 16-bit ALU; however, the bytecodes tend to hide the actual word size from the programmer. The virtual address space is broken down into units of 512-byte pages, and the three machines have different degrees of hardware assist for virtual memory management and instruction fetch.

Cons cells are cdr coded in a manner described in D. Bobrow and D. Clark, "Compact Encodings of List Structure", ACM Trans. on Prog. lang. and Systems, Vol 1 No 2, p266 October 1979. A cell of 32 bits is used to store a cons -- typically 24 bits for the car, and 8 bits for an encoding of the cdr. The encoding covers the four cases where (1) the cdr is NIL, or (2) the cdr is directly on the same page as the cons cell, or (3) the cdr is contained in another cell on the same page as the cons cell, or (4) the cons cell is itself a full indirect pointer, which can address an ordinary two-cell slot on any page (the space normally used for the car is used to address a 64-bit cell elsewhere; this is to allow for RPLACDs when there is no more free cells on the same page as the cell being updated). All cons cells are cdr-coded, independent of how they are created, and as a consequence the "average size" of such a cell is considerably less than 64 bits.

Strings and arrays are implemented as a fixed-length header, with one field pointing to a variable-length memory chunk taken from an area which is separately managed. To run some of the benchmarks, we used Interlisp's Common Lisp array utility package. Additionally, Interlisp permits the user to define new first-class fixed-length data types, with corresponding entries in the quantum map mentioned above; for example, a STREAM is implemented as a record structure with 19 pointer fields and assorted integer fields of 16 bits or less.

Garbage collection is patterened after Deutsch and Bobrow, "An Efficient, Incremental, Automatic Garbage Collector" CACM, July 1976. A reference count is maintained for every collectible pointer (in addition to immediate pointers, litatoms are not reclaimed in Interlisp-D). Updates to non-stack cells in data structures (i.e., the CAR slot of a CONS cell, or the value-cell of a global variable) require updates to the reference count. The reference counts are maintained separate from the objects in a hash table, which is generally very sparse; and the updating is normally done within the microcode that effects the update operations. Reclamations are performed frequently, and involve scanning the stack area and augmenting the reference counts by a "stackp" bit; then scanning the reference count table reclaiming any entry which has a count of 0 and no reference from the stack (and possibly additional pointers whose reference count goes to zero as a result of such a reclamation); and finally re-scanning the table to clear the "stackp" bits. The scan through the reference count table looking for 0-count entries corresponds roughly to the scan of the marked-bits table in a Mark-and-Sweep collector; however, the scan of the stack is infinitesimal in time compared to a full "mark" phase, and thus a reclamation typically runs in well under a second.

The internal architecture of the stack is a variant of the "spaghetti stack" model described in Bobrow and Wegbreit "A Model and Stack Implementation of Multiple Environments, Comm. ACM, Vol. 16, No. 10, Oct. 1973, pp. 591-603. The stack area is currently limited to 128KB.

The particular configurations upon which the benchmarks were run are as follows:

Xerox 1100: (Dolphin). 4K words of 40-bit microstore; microinstruction time 180ns; hardware assist for macro-instruction fetch; hardware memory map for up to 8MB of virtual space; hardware stack (for stack tip); memory access is 1-to-4 words (64 bits) in about 2us. The particular unit used in the benchmarking runs had 1.8MB of real memory attached, but 2MB has been in standard delivery.

Xerox 1108: (DandeLion) 4K words of 48-bit microstore; microinstruction time 137ns; hardware assist for macro-instruction fetch; hardware assist for virtual memory management (memory map is kept in non-paged real memory); memory access is 1 non-mapped 16-bit word in 411ns, but a random 32-bit cell access in about 1.2us. The stack is held in real, non-mapped memory. The particular unit used in the benchmarking runs had 1.5MB of real memory attached.

Xerox 1132: (Dorado) 4K words of 34-bit high-speed ECL microstore; microinstruction time 64ns; hardware instruction fetch unit; hardware memory map for up to 32MB of virtual space; 4Kilowords of high-speed ECL memory cache permit memory access of one 16-bit word in 64ns, and a cache-reload of 256 bits takes about 1.8us (additional details on the cache and memory organization may be found in D. Clark, B. Lampson, and K. Pier: "The Memory System of a High-Performance Personal Computer", IEEE Transactions on Computers, vol C-30, no. 10, Oct 1981). The particular unit used in the benchmarking runs had 2MB of real memory attached.

Note that the benchmarks were not run on the 1108-111 (DandeTiger), which has considerably more memory and control store than the basic 1108, and which also has a floating-point processor.

Interlisp-D
Implementor’s Reference

Filed as: {Eris}<LispCore>Internal>Doc>ImplManual.Tedit;22

Contents

Updating the InterLisp Reference Manual [Sannella]:

Documentation files are kept on {eris}<LispManual>*.im. LispCore members are encouraged to modify these documentation files to reflect the changes made as the system is modified. Please be careful.

One important note: it is necessary to keep track of how the manual is changed, in order to provide a list of changes with the next revision of the manual. Therefore, I would strongly suggest that whenever anyone makes a significant change to the manual (adds/deletes a function definition, adds new arguments to a function, non-trivial rewording, etc.) that they send a short message to me (Sannella, not LispCore^).

The manual is stored in a large number of separate files, and it is difficult to know which file contains a particular function definition. Therefore, I have created a small package that will take an "IM Name" (a function, variable, property name, etc), create a TEdit window on the appropriate IM file, and position the TEdit cursor at the right place.

To use this, do

```
_(FILESLOAD (FROM LISPUSERS) IMNAME)
_(INSPECT.IM 'FOO)
```

`INSPECT.IM` uses the hash file package to search a hash file containing index information for the name FOO. If it is found, it will put up a pop-up menu listing references in different files. Selecting one of the references will move move the cursor in the appropriate TEdit window (if there is an active TEdit window to the appropriate file), or create a new TEdit window to the appropriate file.

Sometimes, a particular name is defined as more than one "type" (function, variable, etc.). In this case, a pop-up menu will prompt you to declare which type you are interested in.

A somewhat more convenient way of using this facility, if you want to use it repeatedly, is to do

```
_(MAKE.IM.INSPECTOR)
```

This sets up an “IM Inspector Window”, which contains a menu. Initially, this contains the single selection "Type an IM name", which prompts the user to type a name which will be looked up in the database. Below this window will appear type-selection and reference-selection menus, which do not disappear until another selection is made above them. This is hard to describe.... try it out. [It works great! -- LMM]

MAKING A LOADUP:

Loadups can currently only be made on Dorados. Command files, with extension "CM", are read by the Alto exec using the "@" command.

There are various command files on {Eris}<LispCore>CM>:

LoadFull.CM makes a LISP.SYSOUT and a FULL.SYSOUT from scratch.

LoadFullFromLisp.CM makes a FULL.SYSOUT directly from the LISP.SYSOUT.

LoadDemoFromFull.CM makes a DEMO.SYSOUT

[Note that the command files were modified to use standard cache partitions on Dorados (which are?). These are used to load the sysout that the renamed functions run in, and to save the sysouts. The old non-cached behavior can be had by using command files whose names begin with SLOW-. A complication this causes is that changes to the command files must now be made in two places, the SLOW- and caching versions.]

The first part of the command file runs MAKEINIT to create a file INIT.SYSOUT and then DLFIXINIT to make it dandelion bootable by merging in the dandelion microcode to create an INIT.DLINIT.

[FS: THE FOLLOWING INDENTED SECTION HAS BEEN CHANGED. THE CORRECT EVENTS FOLLOW BELOW:

> The next part starts up the INIT.DLINIT (which will run on all machines), and calls LOADUP(HUGE). The function LOADUP (on the file APUTDQ which is merged in at MAKEINIT time) has directions on how to do various kinds of loadups. It determines what other files are in the default loadup.
>
> If you have a special kind of loadup that you think should be supported for some applications, it is possible to add a separate clause to LOADUP and include that in the standard source.]

The next part starts up the INIT.DLINIT (which will run on all machines), and the CM script loads LOADUP.LISP and LOADFULL.LISP, which call the function LOADUP on the rest of the files in the LISP.SYSOUT and the FULL.SYSOUT, respectively. (LOADUP 'HUGE) is obsolete and should be deleted.

How MAKEINIT works:

Basically, all the storage-modifying functions are redefined so they make their changes to a file. After some initialization of the blank memory space in the SYSOUT file the normal LOAD code is run, but the effects take place in the new sysout file instead of in memory.

Modification of the low-level storage functions is done by the code in the file RENAMEFNS, based on information in the file FILESETS. The function DORENAME is called with the argument I (for INIT). DORENAME uses the RENAMETYPES variable to determine which files to get low level definitions from, and how to rename them. RENAMETYPES also indicates how to create an R (for READSYS) type rename, which is used by teleraid to read definitions out of a sysout (or sysout file).

The end result of DORENAME I is a file I-NEW, containing the "remoted" definitions. This file is loaded and the "renamed" loaded code is run to load up the earliest parts of the system.

The basic goal of MAKEINIT is to make the INIT.SYSOUT capable of loading files over the ethernet and writing out (as with LOGOUT or MAKESYS) the resulting system. This minimal set of files to load is defined in FILESETS in the variables 0LISPSET and 1LISPSET.

Once these files are loaded, INIT.SYSOUT is written out. After having the dandelion microcode spliced into the memory image by DLFIXINIT the resulting INIT.DLINIT is run. The first thing done here is to run the "init expressions" of all the files which were loaded renamed. The expressions could not be evaluated remotely earlier in the init (indeed, the evaluator is not fully loaded at this point).

Things to watch out for: The data type STREAM must be the first datatype declared after MAKEINIT time. This means that no file loaded before FILEIO can declare a datatype. Packages are "turned off" in the early part of the init and symbols written into the INIT.SYSOUT are all package qualified. The file PACKAGE-STARTUP makes the switchover (located at the end of 1LISPSET).

Now the rest of the files in FILESETS are loaded (those in 2LISPSET and up). The greatest number of problems encountered after this point, aside from outright bugs, are dependencies of code on parts of the system which have not yet been loaded.

After this point the LOADUP function loads in the standard sets of files to make LISP.SYSOUT and FULL.SYSOUT (more detail?).

Writing Renamable Code:

Renamed code is used to build loadups and read the format of the resulting memory spaces (see sections above and below).

This is intended as a start at describing what you need to knwo to write renamable code.

`\COPY` is called to move something from the local memory space to the remote one.

`\UNCOPY` brings objects back.

`LOCAL` can be used to ensure that a function's effects occur in the local memory image. It inhibits renaming of forms inside of it.

`ALLOCAL` can be used to ensure that a function's effects occur in the local memory image. It inhibits renaming of forms inside of it.

`UNLESSRDSYS` takes two forms, the first to execute normally, and the second to be used when the function is renamed.

Since `DTEST` doesn't run renamed, code that is intended to run renamed should use `ffetch` and `freplace`

How Teleraid works:

Teleraid runs in two parts: a *server*, running in microcode on one machine (usually entered by pressing UNDO to a front panel error), and a *front-end*, running on another machine. The front end machine examines the other over the network using the server.

The teleraid server is actually a simple memory page server written in microcode. It transfers pages of memory (using PUP protocols) to the front-end machine. It is up to the front-end machine to understand the internal format of the other machine's memory. This happens through a reverse version of the init process. Rather than renaming functions to write onto a remote memory space (in a file), teleraid runs functions which are renamed to read a remote memory space (on another machine or in a file; you can teleraid a non-running sysout file).

A variation of the renaming scheme used to build the init is employed in Teleraid. A large number of functions in the system are renamed to call (at their lowest levels) the Teleraid page server on the other machine.

Since the lowest levels of the system can change between releases it is important to have the same sysout running on the two machines.

The actual file that contains teleraid's renamed functions is `RDSYS`. It is created automatically by `DORENAME` on the file `RENAMEFNS` and must be updated whenever low level representation is changed. Details of what to rename for teleraid are contained on the file `FILESETS`. The functions which get renamed are scattered all through the low level system files (but those are pointed to by `FILESETS`).

Internal Storage Reference Functions [Masinter, van Melle]:

There are a number of low-level functions for directly accessing memory as if it were an enormous array of 16-bit words, bytes, 32-bit cells, etc. In general, don't call any of these directly if you can help it. They are generally "unsafe" and can confuse your system in subtle ways if misused. Also, there are often alternatives that, if not completely safe, are at least less prone to error:

(A) Write functions or macros to do the accesses, and have them perform suitable type checking. See the `GETPUPWORD` and `PUTPUPWORD` macros, for example.

(B) Define a `BLOCKRECORD` to overlay a given data structure. This is much better than using `\GETBASE` et al if you are using fixed offsets—it is safer (less error-prone) and generally produces better code. With creative use of `LOCF` and `ACCESSFNS`, you can often avoid using explicit `\GETBASE`s altogether, and your code is much more readable. Also check out the `MESATYPES` package, written by Tayloe Stansbury, for producing such expressions from Mesa type declarations.

`(\ALLOCBLOCK` *NCELLS GCTYPE INITONPAGE ALIGN*`)`

The basic low level storage allocation function. *NCELLS* is the number of 32 bit cells to allocate. *GCTYPE* is an integer, usually stated as the value of either UNBOXEDBLOCK.GCT (NIL is an old style synonym), PTRBLOCK.GCT (T is an old style synonym) or CODEBLOCK.GCT. *INITONPAGE* is the number of cells at the beginning of the block which must be allocated on the same page. *ALIGN* is the alignment of the address of this block in memory space. The base address will be evenly divisable by this number.

The argument *BASE* in the following functions refers to an address, an Interlisp pointer. For example, if the value of X is an instance of a datatype, then X is actually a pointer to the first cell of that instance. There are essentially only two operations that perform "pointer arithmetic": `\ADDBASE` and `\VAG2`; these compile directly into the `ADDBASE` and `VAG2` opcodes.

`(\ADDBASE` *BASE OFFSET*`)`

Produces a new address that is *OFFSET* 16-bit words beyond *BASE*.

`(\VAG2` *HI LO*`)`

Produces an address whose left 8 bits is *HI* and whose right 16 bits is *LO*.

There are, however, many other ways to produce addresses that ultimately perform `\VAG2` or `\ADDBASE`, and these are usually preferable. The record `POINTER` is useful for decomposing pointers into page# and word-in-page or cell-in-page quantities. `LOCF` is useful in conjunction with `BLOCKRECORD`s and `DATATYPE`s.

`(LOCF (`**fetch** `FIELDNAME` **of** `datastructure))` [Macro]

Produces a pointer to the first word containing `FIELDNAME`. E.g., if `BAR` is declared as a `WORD` field in a record, then `(fetch BAR of X)` is equivalent to `(\GETBASE (LOCF (fetch BAR of X)) 0)`.

(INDEXF (**fetch** FIELDNAME **of** T)) [Macro]

> Returns the word offset to the first word containing FIELDNAME. Since this is independent of the actual datum being operated on, the datum is often given as "T". E.g., if BAR is declared as a WORD field in a record, then (fetch BAR of X) is equivalent to (\GETBASE X (INDEXF (fetch BAR of T))). There is rarely any need for INDEXF.

Note that \ADDBASE, LOCF and other pointer-producing operations are *not* in general safe in Interlisp-D. The garbage collector can get very confused if you save away arbitrary pointers anywhere other than in local variables. This is because the reference count of an object is associated only with the pointer to its beginning, i.e., only with the address that the public traffics in (the pointer returned from **create**, for example). If you must store away internal pointers, be very careful that you continue to hold on to the pointer to the start of the object for as long as you maintain the internal pointer. This assures that the object will not get garbage-collected out from under you, the most common source of such confusion.

(\ADDBASE2 *BASE N*)

> Equivalent to (\ADDBASE *BASE* 2**N*).

(\GETBASE *BASE OFFSET*)

(\PUTBASE *BASE OFFSET VALUE*)

> These fetch and store, respectively, the 16-bit word (as a Lisp small positive integer) located at *OFFSET* words beyond *BASE*. \PUTBASE is *really* dangerous. E.g., (\PUTBASE NIL n) for many small values of n will smash your system beyond repair. Not good for a residential environment where a smashed system can lose a lot of work.

(\GETBASEBYTE *BASE OFFSET*)

(\PUTBASEBYTE *BASE OFFSET BYTE*)

> Fetch and store 8-bit quanta. *BASE* is a word address, and *OFFSET* is a byte offset—counting the high byte of the base word as offset zero.

(\GETBASEPTR *BASE OFFSET*)

> Fetches a pointer at *OFFSET* from *BASE*. A pointer is a 24-bit quantity, which is stored right-justified in a 32-bit cell. Note, however, that *BASE* and *OFFSET* are both still in terms of 16-bit words.

(\PUTBASEPTR *BASE OFFSET PTR*)

> Stores pointer *PTR* at *OFFSET* from *BASE*. This is not a direct inverse of \GETBASEPTR, because it stores a full 32 bits, never mind what used to be the high 8 bits originally stored there. \PUTBASEPTR does not do reference counting, so this can be especially dangerous if not used carefully. *BASE* is a word address, and *OFFSET* is in *words*, not cells! \RPLPTR is similar to \PUTBASEPTR, except that it *does* do reference counting.

(\RPLPTR *BASE OFFSET PTR*)

> Stores a 24-bit pointer, similar to \PUTBASEPTR, except that (a) it stores only 24 bits, preserving whatever used to be in the high 8 bits; and (b) it does reference-counting operations; decrements the count of the pointer being smashed and increments the count of

the pointer being put into the location. This is the proper way to smash a pointer field if you must. However, there is almost never any need for you to call this directly; the usual way to smash a pointer field is to use records.

Implementation notes: \GETBASEBYTE and \PUTBASEBYTE compile directly into the corresponding opcodes, and execute entirely in microcode when the *OFFSET* and *VALUE* arguments are small positive integers. \GETBASE, \PUTBASE, \GETBASEPTR, \PUTBASEPTR, and \RPLPTR compile directly into the corresponding opcodes when the *OFFSET* argument is a constant less than 256; for other *OFFSET* arguments (variable quantities, larger integers), they require an ADDBASE in addition.

(\GETBASEFIXP *BASE OFFSET*)

(\PUTBASEFIXP *BASE OFFSET VALUE*)

These fetch and store 32-bit integers.

(\GETBASEFLOATP *BASE OFFSET*)

(\PUTBASEFLOATP *BASE OFFSET VALUE*)

These fetch and store 32-bit floatps.

(\GETBASESTRING *BASE OFFSET NCHARS*)

Creates a string *NCHARS* characters long whose characters consist of the bytes located starting at *OFFSET* (a byte offset) from *BASE*. Thus, the first character of the result is (\GETBASEBYTE *BASE OFFSET*).

(\PUTBASESTRING *BASE OFFSET STRING*)

Stores the characters of *STRING* as consecutive bytes starting at *OFFSET* (a byte offset) from *BASE*.

(\BLT *DBASE SBASE NWORDS*)

Copies a sequence of *NWORDS* words starting at *SBASE* to corresponding words starting at *DBASE*. This compiles directly into the BLT opcode. In the case where the source and destination ranges overlap, the behavior is well-defined: words are copied from the end of the range backwards to the beginning. Thus, this is equivalent to (for I from NWORDS-1 to 0 by -1 do (\PUTBASE DBASE (\GETBASE SBASE I))). Very fine point: this operation is defined to be completely uninterruptable if NWORDS is less than 10; thus, you can use opcode to make small indivisible transfers.

(\MOVEWORDS *SBASE SOFFSET DBASE DOFFSET NWORDS*)

Obsolete predecessor of \BLT.

(\MOVEBYTES *SBASE SBYTEOFFSET DBASE DBYTEOFFSET NBYTES*)

Copies a sequence of *NBYTES* bytes starting at *SBYTEOFFSET* bytes beyond *SBASE* to *DBYTEOFFSET* bytes beyond *DBASE*. If the ranges overlap, the result is formally undefined.

(\ZEROBYTES *BASE FIRST LAST*)

Stores zeroes into the bytes at offsets *FIRST* thru *LAST*, inclusive, from *BASE*. Thus, a total of *LAST-FIRST*+1 bytes are cleared.

(\ZEROWORDS *BASE ENDBASE*)

Stores zeroes into the words from *BASE* thru *ENDBASE*, inclusive. There are obscure reasons for the lack of symmetry among \ZEROWORDS, \ZEROBYTES, and \MOVEBYTES.

ABC, EXPORTS.ALL, &c:

Note that for Interlisp-D, LOAD(ABC) loads <LispCore>Library>EXPORTS.ALL.

(LOAD 'MAKE-EXPORTS.ALL) will connect to <LISPCORE>SOURCES> and gather all exports into the EXPORTS.ALL file.

"ABC" stands for "A Byte Compiler"—meaning the augmented environment required to compile Interlisp system code. The augmentation includes any of the definitions found under the EXPORT FILEPKG command. The variable EXPORTFILES, set up by loading the file FILESETS, contains the rootname of all system files which have any EXPORT commands. A file wil generally export those items that other files need (e.g., records or macros) which are DONTCOPY, and thus not part of the user's system.

SYSRECORDS [van Melle]:

In order for the inspector to be able to inspect an object of some user-declared datatype, it needs a declaration for it. The declarations for system datatypes are omitted from the loadup (by being marked DONTCOPY and being initialized with INITRECORDS). In order for their instances to be inspectable, they should be added to SYSTEMRECLST by the filepkg command SYSRECORDS, which is syntactically identical to RECORDS. The datatype declaration is actually stripped of comments, subrecords and initialization info before being put out.

Putting new DLion Microcode into a Sysout:

Load the file `SPLICE.DCOM` from <Lispcore>Sources>.

(`NCLIP` *MICROCODEFILE SYSOUTFILE*)

Copies the entire contents of *MICROCODEFILE*, a DLion .db file, into *SYSOUTFILE*, which must be a Lisp sysout. Both files must be random access.

DTigerness vs. DLionness

How programs can tell if they're running on a DTiger

Programs can tell if they are running on a Dandetiger (1108 with CPE) by evaluating `(AND (EQ \MACHINETYPE \DANDELION) (ODDP (\DEVICE.INPUT 8)))` - this will return T when run on a DTiger, NIL when run on anything else. `(\DEVICE.INPUT 8)` returns the microcode version number when run on an 1108; if the version is odd, you're running on a DTiger.

WHEREIS, and the WHEREIS database

The WHEREIS package is used to find where (which source package) an atom has come from.

The expanded WHEREIS package comes preloaded in FULL.SYSOUT, or can be loaded from {ERIS}<LISPCORE>LIBRARY>WHEREIS.DCOM. Normally WHEREIS only searches loaded files. If the function WHEREIS is called with a FILES argument of T it searches the list of hashfiles given in the global WHEREIS.HASH. These hashfiles may have names such as WHEREIS.HASH, LIBRARY.WHEREISHASH, SYSTEM.WHEREISHASH, etc., and may live in places like {ERIS}<LISP>INTERMEZZO>LIBRARY. Usage:

```
_(WHEREIS <foo> 'FNS T)
```

This definition also allows you to ask for MACROS, RECORDS, PROPS, and VARS in addition to FNS.

To make a new WHEREIS database use the function WHEREISNOTICE thus:

```
_WHEREISNOTICE(
      (<LISPCORE>SOURCES <LISPCORE>LIBRARY> <LISPUSERS>)
      T
      <LISPCORE>SOURCES>SYSTEM.WHEREISHASH]
```

Note that it takes a long time to examine all the files. Best to leave this task to a lonely Dorado.

Installing New Opcodes [Masinter]:

[abstracted from messages to Jim desRivieres]

The various functions (CALLSCCODE, PRINTCODE, CHANGENAME) know about opcodes via the list \OPCODES. If you want to add some new opcodes, you can edit the list. The format of \OPCODES is documented, I think, in the function PRINTOPCODES, which is on the file ACODE. Note that if you install new opcodes on the fly you should then reset the variable \OPCODEARRAY to NIL.

You can tell if you have installed the opcodes by calling \FINDOP directly. (\FINDOP 'CAR) should return the opcode-description-record for CAR, while (\FINDOP 231Q) should look up opcode 231.

If you add opcodes, you should send a message to LispCore^ outlining what opcodes you want to reserve. The file OPCODES.TEDIT (on <LispCore>INTERNAL>DOC>) I think has a listing of opcodes too, and if you are reserving a range, that reservation should be documented there too.

Subject: adding UFNs

The UFN mechanism hasn't really been extended for simple experimentation but is workable with a little effort. Normally, UFN entries get set up at MAKEINIT time by a renamed version of a function, I think it is called \SETUFNENTRY or some such. (LLCODE, LLBASIC, LLNEW or one of those). The entries in the OPCODES record is used to set up the ufns. Now, it is currently the case that UFN's can't do anything like push N things on the stack -- all they can do is pop N arguments (N>=0) and push 1 result.

Writing ufn's that do something other than that, e.g., that don't follow the normal function call paradigm, are a lot more work. Basically I think you have to get into the level of stack-hacking that is found inside LLSTK. For example, a UFN that wanted to push a bunch of NIL's would have to do something awful, like steal space out of its own basic frame to give it back to the caller. This kind of code is tricky to write and debug, especially because you can't do things like insert BREAKs.

Popping N off of course is easy since that is what function calls do.

In order to do a jump operation, doing something like (add (fetch PC of (\MYALINK)) 10) would do a relative jump to byte +10.

It may actually be necessary to extend the UFN mechanism to allow some of the extensions that you want. Why don't you figure out what you can do with the current mechanism, and come back with the ones that you can't figure out how to implement.

Adding new opcodes to the Interlisp-D system

Written by: Herb Jellinek
Revised: 14 June 1984

The process of adding new opcodes to the Interlisp-D system has long been a mysterious one. This document is an attempt to shed some light on these mysteries. The document covers: Creating new opcodes, Writing UFNs, and The OpcodeTool. Enjoy.

Creating new opcodes

There are a number of global objects and properties that one must know about in order to install new opcodes/UFNs. Here's a list of them:

`\OPCODES` [List]

A list of the current opcodes, each of which is a record of type `OPCODE`.

`\OPCODEARRAY` [Array]

An array-ified version of `\OPCODES`. If set to `NIL` it will be reinitialized from the contents of `\OPCODES`. `\OPCODEARRAY` is recreated when needed by the function `\FINDOP`.

`DOPVAL` [Property]

Information on how to emit code for a given function. There are two formats:

1. (*nargs* . *opcode-sequence*)

If the number of arguments supplied matches *nargs*, compile into the sequence *opcode-sequence*.

2. ((*nargs1* . *opcode-sequence1*)
   (*nargs2* . *opcode-sequence2*) ...
   (*nargsN* . *opcode-sequenceN*) . *other-cases*)

If the number of arguments supplied matches *nargs1*, compile into *opcode-sequence1*, otherwise see if the number of arguments supplied matches *nargs2*, etc. One may also supply a function name as the tail of the list; the code generator will call that function if none of the other cases apply. The function `OPT.COMPILERERROR` is typically used for this purpose; it is equivalent to `HELP`.

`DOPCODE` [Property]

The `OPCODE` record for a given atom.

`OPCODE` [Record]

A record describing the structure of the `DOPCODE` property and the elements of the list `\OPCODES`; it has the following fields:

UFNFN	name of the ufn. Actually read by MAKEINIT
LEVADJ	stack effect (+/-n) or token. used by PRINTCODE. See code in PRINTCODE for details
OPPRINT	used only by PRINTCODE.
OPNARGS	number of extra bytes
OPCODENAME	name of opcode
OP#	number of opcode, or range for opcode sequences

Herb will document OPPRINT.

Writing UFNs

UFNs are Lisp functions that either run in the place of unimplemented instructions or when the microcode detects a situation that is too complex for it to handle. (This is termed *punting*.) There are two cases involved in writing UFNs: those for single-byte opcodes, and those for multi-byte opcodes.

UFNs for single-byte opcodes

For example, assume we have a single-byte opcode called SQRT, which takes a FLOATP as operand and returns its square root. The instruction is designed to punt out to its UFN (named \SQRT) when its operand is of the wrong type, at which time the UFN can either attempt to coerce the operand to a FLOATP or signal an error. \SQRT need be no more than a function of a single argument.

UFNs for multi-byte opcodes

These UFNs are slightly more complicated, but not much. The difference between single-byte UFNs and multi-byte ones is in the handling of the "extra" (alpha, beta, gamma) byte or bytes. To wit: all multi-byte opcodes that begin with the same byte have the same UFN, and the extra bytes get passed to this UFN in the form of extra arguments. We might have a group of three bit-vector operators (we'll call them BITOP), that all begin with a bytecode of 72Q and vary from 0 to 2 in the second byte. The bytecodes each expect one argument on the stack. The UFN (\BITOP), would probably have the following form:

```
(DEFINEQ
    (\BITOP
        (LAMBDA (BITVECTOR OP)
            (SELECTQ OP
                (0 (\BITOP.MASK BITVECTOR))
                (1 (\BITOP.SHIFT BITVECTOR))
                (2 (\BITOP.ROTATE BITVECTOR))
                (HELP "\BITOP - illegal operation" OP)))))
```

The OpcodeTool

[lmm: I changed the opcode format; I don't know if this works]

[hdj: it doesn't]

[jop: fixed 5-2-86]

The OpcodeTool is a package that makes it easy to set up and test new opcodes in a running Lisp system. Do

```
(LOAD '{Eris}<LispCore>Misc>OpcodeTool.dcom)
```

This package has one entry point, `MAKEOPCODE`, a function which takes 8 arguments:

OPNAME	a litatom
NUM	the opcode number
OPNARGS	the number of extra bytes (alpha, beta, etc.)
OPPRINT	usually T
LEVADJ	the stack level adjustment for this opcode
UFNFN	the UFN for this opcode
UNIMPL	a list describing which machines have no microcode for this op
DOPVAL	a (optional) DOPVAL prop for the litatom OPNAME

After you've run `MAKEOPCODE`, you can compile functions that use the new opcode and test them out.

Creating New Devices:

Date: 15 MAR 84 09:47 PST
From: MASINTER.PA
Subject: AR for Implementors Manual
To: LispSupport
cc: Kaplan, LispCore^

Section on making new file devices, How To.

VMEM.PURE.STATE [van Melle]

When preparing a demo, it is often nice to set things up in such a way that you can push the boot button at any time to instantly restart the sysout, rather than having to go back to some installation utility to reinstall the sysout. `VMEM.PURE.STATE` is a hack that lets you do this. Basically, while it is on, the page fault handler is altered to write dirty pages beyond the original end of the vmem, thus keeping the original vmem "pure".

`(VMEM.PURE.STATE` *`FLG`*`)` [Function]

> When *FLG* is true, enables "pure vmem" as of the next operation that writes out a consistent vmem, e.g., `LOGOUT`, `SAVEVM`, or `SYSOUT`. While in this state, as long as you do not perform another vmem-writing operation (`LOGOUT`, etc.), you can boot the machine (or slightly more cleanly, call `(LOGOUT T)`) and be back in the same state as the `LOGOUT` (or whatever) that initiated the pure image.
>
> When *FLG* is `NIL`, returns to normal page fault operation. This is usually not too interesting, unless you really do want to `LOGOUT`, etc and forgo the "checkpoint" you set up. Note, however, that in either case, your virtual memory file is bloated by whatever pages had been written to the end of the vmem file instead of where they belonged.
>
> There is a mode in which LOGOUT compresses the pages back to where they belong, but I never got it fully debugged.

There is a new MP error in this state: 9316. It means you wrote out so many dirty pages, you ran into the absolute end of the vmem file (8MB) even though you still have plenty of "virtual memory" left.

There are two typical modes of operation:

(1) Call `(VMEM.PURE.STATE T)` before calling `SYSOUT`, thus making a sysout that has the pure feature turned on for anyone running it.

(2) Start up a sysout not so made, and then call

```
(PROGN (CLEARW (TTYDISPLAYSTREAM))
       (VMEM.PURE.STATE T)
       (LOGOUT))
```

to turn the vmem into which this sysout was loaded one with the pure property.

Dorado Mufflers & Manifolds

Date: 23 Sep 84 19:29 PDT
From: JonL.pa
Subject: [JONL.PA: Mufflers and Manifolds]
To: Jellinek

A couple weeks ago I promised to send you "all my knowledge" about the alpha bytes of the MISC opcodes. Not much knowledge; don't even know for sure why I picked "9", but possibly Bill will know of an "8" used for Dolphin-specific purposes.

-- JonL --

----- Begin Forwarded Messages -----

Date: 6 MAR 84 18:42 PST
From: JONL.PA
Subject: Mufflers and Manifolds
To: vanMelle, Charnley
cc: Purcell, JonL

Sorry I didn't mention this before -- after talking with Bill and Don I decided to use opcode MISC1 with alphabyte of 9 to do the equivalent of Mesa's ReadWriteMufflerManifold operation. No MISC opcodes at all were implemented on the Dorado (before now), and the set of alphabytes from 0 through 8 seem to be exhausted by the Dlion/Dolphin needs.

The interface to this operation is:
Input -- one 16-bit integer; low-order 12 bits are Muffler/Manifold address, bit 2^15 non-zero means to write manifold, zero means to read muffler.
Output -- for write operation: NIL. for read operation: one 16-bit integer whose low-order 15 bits are garbage and bit 2^15 is the 1-bit value of the muffler.

See the file <LispCore>Misc>MAKEDORADONSHOSTNUMBER for the code I'm about to install modulo the following: (1) SELECTQ on \MACHINETYPE, (2) don't do it if (MICROCODEVERSION) is less than 12004Q, (3) do "replaces" into (IFPAGE NSHost[i]) rather than cons up the NSHOSTNUMBER record.

----- End Forwarded Messages -----

The DLion Low Level Disk Drivers

Last revised: 26-Jun-84 23:13:59 by Mitch Lichtenberg

This file is an attempt to explain the operation of the Dandelion rigid disk interface, the microcode, and how Lisp constructs and uses disk IOCBs to perform disk operations.

DISK DRIVE INTERFACE

The Dandelion's central processor divides its time among the high speed I/O devices: the ethernet, the rigid disk, the I/O processor, and the display. The "I/O Page" is located in a well-known (to the microcode and Lisp) area of virtual memory, and it holds locations for communication between the different "micro-tasks."

Currently, the Dlion's Disk IOCB is the second word on the I/O page (that is, `(\ADDBASE \IOPAGE 1)`). Memory locations placed on this page must be up to 16 bits long, which constrains the address to be within the first 256 pages.

The IOCB page is used to store parameters and other information that is picked up by the microcode. When one wants to initiate an I/O operation, it can be done by depositing the parameter block onto the IOCB page (somewhere) and then placing the location (16 bits) of the parameter block onto the device's "mailbox" on the I/O page. When the device notices that something has been deposited onto its special I/O page location, it will read in the parameters, execute the operation, and reset the flag to zero to indicate that the operation is complete. (*Note: This is not completely true for the disk.). So, device I/O in Lisp usually looks like the following:

```
(\BLT (\ADDBASE \IOCBPAGE IOCBDisplacement) IOCB IOCBLen)
... or an alteration of an existing IOCB on the IOCB page ...
(\PUTBASE \IOPAGE DeviceCSBDisplacement
       (\LOLOC (\ADDBASE \IOCBPAGE DeviceIOCBDisplacement)))
(until (ZEROP (\GETBASE \IOPAGE DeviceCSBDisplacement)))
```

The `\PUTBASE`s and `\GETBASE`s usually come in the form of record package macros.

The reason why the disk does not follow the usual `\IOPAGE` convention of resetting its CSB to zero is because the drivers were meant to cause interrupts when disk I/O is finished. Lisp does not currently utilize this feature, so to poll the IOCB to detect when it has been completed, the IOCB status is set with some unused bits activated, and when the IOCB completes, the disk microcode will fill in the status and wipe out those bits as a side effect.

DISK IOCBS

To make life easier on the Dandelion's disk microcode, the implementors thought it to be a good idea to have the microcode emulate some kind of primitive "instruction set." So, when you want disk I/O, you have to write a little "program" in "IOCB Machine Language" to accomplish it.

Fortunately, these IOCBs are still around after booting, and Lisp leaves them in place but changes the fields to do more complicated operations.

The entire IOCB page is divided into three sections:

1. The Data Field:

This portion of the IOCB page contains mostly scratch space for the microcode and information for the programmer. Of particular interest here are the Header and Label "template" fields. For read operations, these fields are modified by the microcode, whereas on verify and write operations, they are just read.

The header field corresponds to the header records that were written on the disk when it was formatted. They contain identification information for the microcode - the sector's track, cylinder, and head numbers. These records are *never* written to. The usual operation on this field is verify, and it is primarily used to indicate to the microcode which sector is which on a given track, and to provide a security mechanism for the microcode. This template is also used to store the current cylinder number for the disk drive, and it is the place where the cylinder, track, and sector numbers are stored prior to a disk operation.

The label field is more general-purpose. In the pilot world, the ID number of a file and the page's relative location within the file are stored in the label field. For booting, a coded pointer is also stored here to lead the boot microcode from one sector to the next.

Note that the header and label fields *must* be in these locations on the IOCB page. (they may not be somewhere else in VM). The pointers to these fields from within the parameter blocks are only 16 bits long, so the headers and labels must be kept here.

2. The Parameter Area

The second portion of the IOCB page is the parameter areas for the IOCB programs. There are two of these IOCB parameter areas left over after booting, but Lisp only uses one of them. The information contained in the parameter block includes the run length, the type of operation to use (which operation, read/write/verify, to use on each field), the virtual page number of the disk buffer, and information on how to handle errors. They must be aligned on 16 word boundaries, due to the way that they are loaded into the micro-registers inside the CP.

3. The IOCBs

The third portion of the IOCB page is contains the actual IOCBs themselves. There are two basic types of IOCBs:

1. Seek IOCB:

The Seek IOCB is complete as it stands. (it has no parameter areas to read in like the transfer IOCB has). The Lisp code fills in the fields for the number of cylinders and the direction to seek, and the IOCB's code steps the drive head in the given direction for the given number of steps. There are no verify operations on seeks, so it is the programmer's responsibility to remember which track number the head is currently positioned at. If a disk drive gets lost, a recalibrate operation is necessary. This can be accomplished by setting up an IOCB to step out one track, and continually running it until the `Track00` bit of the disk controller status register becomes T. This register can be read with the function `(\DEVICE.INPUT 3)` and the fields can be found in the record `DLDISK.STATUS` on `DISKDLION`.

2. Transfer IOCB:

The Transfer IOCB reads in a parameter area of 17 words, executes the transfer operation, and exits.

DISK IOCB MACHINE LANGUAGE

As was mentioned before, the disk microcode emulates a very small instruction set (to keep the code size down and increase flexibility). This instruction set is as follows:

Opcode	Operation
`8000 xxxx`	Send word `xxxx` to disk controller register `KCtl`.
`0007 ssss`	Set status bits from `ssss`
`0000 aaaa`	Increment number in location `aaaa`, and skip if zero.
`0002 aaaa`	Unconditional jump to location `aaaa`.
`0006`	Finish up IOCB.
`0400 aaaa`	Write status to location `aaaa`.
`0005 aaaa`	Load parameter table from locations starting at `aaaa`.
`0800`	Transfer a run of pages, skip of no error

Some interesting "`8000 xxxx`" commands follow:

Opcode	Operation
`8000 0422`	Wait for pending seeks to complete. (InsureSeekComplete)
`8000 0420`	Seek step IN (positive direction)
`8000 04A0`	
`8000 0460`	Seek step OUT (negative direction)
`8000 04E0`	

Inside the parameter areas, the following "code numbers" are important:

Code	Meaning
`001E`	Abort on NotReady, WriteFault, Overrun, or CRC errors
`001C`	Abort on NotReady. WriteFault, Overrun
`001F`	Abort on NotReady, WriteFault, Overrun, CRC, or Verify

OTHER NOTES, RESTRICTIONS, ETC.

To specify the length of the data field, "`8100`" is used instead of "`0100`". Setting the high bit of the data length field causes the microcode to increment the virtual page number after each page is transferred. This is used primarily for multiple page runs.

The length of the header and label fields must be decremented for verify operations.

It is impossible to follow a write operation with anything other than a write operation. That is, if you write the label field you *must* write the data field. Otherwise, the tail of the label write operation will trash the data field. (Something in the microcode or disk controller causes this, and it cannot be avoided!)

The files `[Eris]<Lispcore>Dlion>DiskBootIOCBs.bravo` and `[Eris]<Lispcore>Dlion>DiskTest.dm` contain many sample IOCBs. They are invaluable anyone tinkering with the Dlion disk system.

THE LISP DLION DISK HEAD

The heads for the DLion disk are stored on the file `DISKDLION` in the sources directory. The following is a description of the functions in this file and their purposes:

`(\DL.DISKINIT)` [Function]

Determines the shape of the disk drive and sets up variables as follows:

```
\DISKTYPE:     One of \SA4000, \SA1000, \Q2040, \Q2080
SEC/HD:        Sectors per head on this disk drive
SEC/CYL:       Sectors per cylinder on this disk drive
```

The data for each drive follows:

Drive	Sec/Hd	Sec/Cyl	Heads
SA4000	28	224	8
SA1000	16	64	4
Q2040	16	128	8
Q2080	16	112	7

`(\DL.RECALIBRATE)` [Function]

Attempts to find track zero of the disk drive by repeatedly stepping the drive out and checking the status word for `Track00` indication. If more than 512 steps are made and `Track00` still is not true, a call to `RAID` is made.

`(\DL.DISKSEEK CYL)` [Function]

Seeks disk drive to cylinder `CYL`, and updates information in the header template.

`(\DL.TRANSFERPAGE DA BUFFER MODE LABEL RUNLENGTH NORAIDFLG)` [Function]

"User" entry (that is, DLion file system entry) to the disk head. `DA` is the disk address, which may be a fixp. The remaining args are the same as those for `\DL.XFERDISK`, as described below:

`(\DL.XFERDISK CYL HD SEC BUFFER MODE LABEL RUNLENGTH NORAIDFLG)` [Function]

Starts a Disk I/O operation. The argument format is meant to be compatible with the old `\DL.XFERDISK`. This minimizes the confusion with changing the swapper. New features include the ability to work with labels and an error recovery mechanism. If a disk error occurs, the `\DL.XFERDISK` function will retry the operation up to ten times. If it fails, it will do a `(\DL.RECALIBRATE)` first and try ten more times before finally calling `RAID`. Arguments are as follows:

`CYL` Cylinder number of disk address

`HD` Head number of disk address

`SEC` Sector number of disk address

Note: These numbers will be normalized automatically. For example, it is permissible to transfer Cylinder 0, Head 440, Sector 1215. `\DL.XFERDISK` will change that into a meaningful value. This is how the swapper works - see below.

`BUFFER` A pointer to the first `page` that will be used in the disk operation. **Note:** The page must be *locked down*, *touched* (referenced), and *dirty*, or else the swapper will not perform properly!

`MODE` One of the following:

`NIL`	Read pages , read labels (`VRR` operation)
`T`	Write pages, read labels (`VRW` operation)
`VRR`	Read pages, read labels
`VVR`	Read pages, verify labels
`VVW`	Write pages, verify labels
`VWW`	Write pages, write labels
`VRW`	Write pages, read labels (used by swapper)

`LABEL` A pointer to the label record (10 words), or `NIL` if you don't want to use a label record. The label must be locked down to prevent page faults inside the `\DL.XFERDISK` routine.

`RUNLENGTH` The number of consecutive pages to transfer, or `NIL` for one. There are restrictions on multiple page runs: To do a multiple page run, the virtual page numbers of the buffer pages must be sequential, and the run may not cross cylinder boundaries. (See function `\CYLBOUNDCROSSP` below).

`NORAIDFLG` Normally, `\DL.XFERDISK` will bomb after failing to do an I/O operation ("failing" does not include verify errors). (It will call `RAID`). To supress this, set `NORAIDFLG` to `T` and disk errors will be returned as status to the caller.

`(\CYLBOUNDCROSSP DA1 DA2)` [Function]

Predicate returns T if DA1 and DA2 are on different cylinders, NIL otherwise. Note: This function is not locked down.

`(\DL.DISKOP IOCB)` [Function]

Passes IOCB to the disk microcode (which starts the I/O operation), waits for it to complete, and returns the status.

`(\D2V CYL HDSEC)` [Function]

Returns the disk address of the page on cylinder CYL and with encoded head and sector information in HDSEC (left byte is head number, right byte is sector number)

`(\V2HDSEC DA)` [Function]

Returns encoded head and sector information from disk address.

`(\V2CYL DA)` [Function]

Returns cylinder number from disk address.

`(\DL.ACTONVMEMPAGE FILEPAGE BUFFER WRITEFLG)` [Function]

Performs a file operation on the virtual memory file. `FILEPAGE` is a file relative page number to transfer, `BUFFER` is the page number for the transfer, and `WRITEFLG` is passed to `\DL.XFERDISK` as the `MODE` parameter. It is usually `T` or `NIL`. This function is implemented by figuring the starting address triple of the beginning of the VMEM file and computing the number of pages into the disk from there that the page is located (skipping bad pages), then supplying this information as the sector number to `\DL.XFERDISK`, which normalizes it internally to a real disk address.

`(\DL.ACTONVMEMFILE FILEPG BUFFER NPGS WRITEFLG)` [Function]

Performs multiple file operations on the virtual memory file. `FILEPG` is the starting file page number (relative to the start of the VMem file). `BUFFER` is a pointer to the first page in the group to be transferred. NPGS is the number of pages to transfer. `WRITEFLG` passed to `\DL.ACTONVMEMPAGE` as the `WRITEFLG` parameter. This function will transfer a run of pages to or from the virtual memory file.

`(\DLDISK.GETSTATUS)` [Macro]

Returns the status of the disk controller in a smallp. Use the record definition `DLDISK.STATUS` to understand its contents. This macro expands to `(\DEVICE.INPUT 3)`

DLDISK.STATUS [Record Definition]

Record defintion (access functions) for reading the result of (\DLDISK.GETSTATUS). Contains the followng fields:

TRACK00	True if on track zero
HEADSELECT	Curreny head number
SA1000	True if controller is in SA1000 mode
DRIVENOTREADY	True if drive is not ready
WRITEFAULT	True if last operation caused a write fault
OVERRUN	True if last operation caused an overrun
CRCERR	True if last operation caused a CRC error
VERIFYERR	True if last operation caused a verify error

IOCBPAGE [Record Definition]

This record contains the layout of the IOCB page. The fields are as follows:

LASTIOCBSTATUS	Last status reported while running IOCB
NEXTIOCG	Short pointer to next IOCB in chained IOCBS. This is not currently used.
SEEKIOCBLOC	Contains the location of the SEEK IOCB.
XFERIOCBLOC	Contains the location of the TRANSFER IOCB
VRRIOCBLOC	Contains the location of the VRR Parameter block
VVRIOCBLOC	Contains the location of the VVR Parameter block
HCYLINDER	Header Template: Contains current cylinder number. Changed in all operations.
HHEAD	Header Template: Contains current head number. Changed in all operations.
HSECTOR	Header Template: Contains current sector number. Changed in all operations.
LID	Label Template: 5 words of ID number for the label.
LPAGELO	Label Template: Low 16 bits of page-within-file information in the label.
LPAGEHI	Label Template: High 7 bits of page number within file.
LFLAGS	Label Template: Flag storage for boot code
LTYPE	Label Template: Type of page (type of file in which the page is a part) (16 bits)
LBOOTLINKCHAIN1	Label Template: Boot chain info
LBOOTLINKCHAIN2	Label Template: Boot chain info
PRUNLENGTH	Parameter Block: Run length (number of pages)
PLABELCMD	Parameter Block: Code for operation on label field
PLABELLEN	Parameter Block: Length of label field
PLABELABORT	Parameter Block: Conditions for aborting transfer & error codes to scan for
PDATACMD	Parameter Block: Code for operation on data field
PDATALEN	Parameter Block: Length of data field
PVPAGE	Parameter Block: Virtual page number of memory buffer

PDATAABORT	Parameter Block: Conditions for aborting transfer & error codes to scan for
PTERMCOND1	Code to halt hardware after transfer
PTERMCOND2	Code to halt hardware after transfer
SCYLINDERDISPLACEMENT	Seek IOCB: Number of cylinders to move in seek operation.
SSEEKCMD1	Seek IOCB: First part of seek command
SSEEKCMD2	Seek IOCB: Second part of seek command

DISK IOCB PAGE

This section contains the contents of the IOCB page.

Displacements are relative to the start of the IOCB page.

Lines with asterisks following the opcode indicate fields for the user to fill in.

Address Op/Data Comment

```
0100:       000B         ; Special Block Type
0101:       00FE         ; Word count
0102:       0000         ; Not used
0103:       0000         ; IOCB Status (filled in by uCode)
0104:       0000   *     ; Next IOCB address (or 0 for last one)
0105:       0120         ; Address of seek IOCB
0106:       0132         ; Address of transfer IOCB
0107:       0140         ; Address of verify-read-read parameter area
0108:       0160         ; Address of verify-verify-read parameter area
0109:       0180         ; Address of verify-verify-write parameter area
010A:       01A0         ; Address of verify-write-write parameter area

010B:       0000   *     ; Header template: Cylinder number
010C:       0000   *     ; Header Template: Head[0..7], Sector[0..7]

010D:       0000   *     ; Label Template: Word 0 \
010E:       0000   *     ; Label Template: Word 1  \
010F:       0000   *     ; Label Template: Word 2   > ID Number for page
0110:       0000   *     ; Label Template: Word 3  /
0111:       0000   *     ; Label Template: Word 4 /
0112:       0000   *     ; Label: Page # low [bits 7..22]
0113:       0000   *     ; Label: [Pg# Hi 0..6, Pad 7..12, Flags 13..15]
0114:       0000   *     ; Label: Type #
0115:       0000   *     ; Label: Unused
0116:       0000   *     ; Label: Unused
0117:       0000         ; Filler for 16 wrd boundary lineup
0118:       0000         ; Filler for 16 wrd boundary lineup
0119:       0000         ; Filler for 16 wrd boundary lineup
011A:       0000         ; Filler for 16 wrd boundary lineup
011B:       0000         ; Filler for 16 wrd boundary lineup
011C:       0000         ; Filler for 16 wrd boundary lineup
011D:       0000         ; Filler for 16 wrd boundary lineup
011E:       0000         ; Filler for 16 wrd boundary lineup
011F:       0000         ; Filler for 16 wrd boundary lineup
```

Parameter Area for Verify-Read-Read IOCB

```
0140:        0000   *      ; Number of sectors to read
0141:        0031          ; Max #+1 of secs that may be skipped searching
0142:        0432          ; Verify header field
0143:        0001          ; word count-1 of header field
0144:        010B          ; address of header field
0145:        001C          ; Abort on NotReady. WriteFault, Overrun
0146:        0003          ; skip to next sector if CRC/Vrfy err on header
0147:        0430          ; Read label field
0148:        000C          ; word count of label field
0149:        010D          ; Address of label field in IOCB
014A:        001E          ; Abort on NotReady, WriteFault, Overrun, CRC
014B:        0430          ; Read Data Field
014C:        8100          ; Length of data field (256 words)
014D:        0000   *      ; virtual page # of buffer
014E:        001E          ; Abort on NotReady, WriteFault, Overrun, CRC
014F:        0420          ; control word to halt hw after each field
0150:        0426          ; control word to find sector mark for header
```

Parameter Area for Verify-Verify-Read IOCB

```
0140:        0000   *      ; Number of sectors to read
0141:        0031          ; Max #+1 of secs that may be skipped searching
0142:        0432          ; Verify header field
0143:        0001          ; word count-1 of header field
0144:        010B          ; address of header field
0145:        001C          ; Abort on Not Ready. Write Fault, Overrun
0146:        0003          ; skip to next sector if CRC/Vrfy err on header
0147:        0432          ; Verify label field
0148:        000B          ; word count of label field (-1 for verify)
0149:        010D          ; Address of label field in IOCB
014A:        001F          ; Quit on NotRdy, WrtFlt, Ovrrn, CRC, Verif Err
014B:        0430          ; Read Data Field
014C:        8100          ; Length of data field (256 words)
014D:        0000   *      ; virtual page # of buffer
014E:        001E          ; Quit on NotReady, WriteFault, Overrun, or CRC
014F:        0420          ; control word to halt hw after each field
0150:        0426          ; control word to find sector mark for header
```

Parameter Area for Verify-Verify-Write IOCB

```
0140:        0000   *      ; Number of sectors to read
0141:        0031          ; Max #+1 of secs that may be skipped searching
0142:        0432          ; Verify header field
0143:        0001          ; word count-1 of header field
0144:        010B          ; address of header field
0145:        001C          ; Abort on Not Ready. Write Fault, Overrun
0146:        0003          ; go to next sector if CRC/Verfy err on header
0147:        0432          ; Verify label field
0148:        000B          ; word count of label field (-1 for verify)
0149:        010D          ; Address of label field in IOCB
014A:        001F          ; Stop on NotRdy, WrtFlt, Ovrrn, CRC, Verif Err
014B:        043B          ; Write Data Field
014C:        8100          ; Length of data field (256 words)
014D:        0000   *      ; virtual page # of buffer
014E:        001C          ; Abort on NotReady, WriteFault, Overrun
014F:        0420          ; control word to halt hw after each field
0150:        0426          ; control word to find sector mark for header
```

Parameter Area for Verify-Write-Write IOCB

```
0140:        0000   *      ; Number of sectors to read
```

```
0141:       0031          ; Max #+1 of secs that may be skipped searching
0142:       0432          ; Verify header field
0143:       0001          ; word count-1 of header field
0144:       010B          ; address of header field
0145:       001C          ; Abort on Not Ready. Write Fault, Overrun
0146:       0003          ; skip to next sector if CRC/Vrfy err on header
0147:       043B          ; Write label field
0148:       000C          ; word count of label field
0149:       010D          ; Address of label field in IOCB
014A:       001C          ; Abort on Not Ready. Write Fault, Overrun
014B:       043B          ; Write Data Field
014C:       8100          ; Length of data field (256 words)
014D:       0000    *     ; virtual page # of buffer
014E:       001C          ; Abort on NotReady, WriteFault, Overrun
014F:       0420          ; control word to halt hw after each field
0150:       0426          ; control word to find sector mark for header
```

Parameter Area for Verify-Read-Write IOCB

```
0140:       0000    *     ; Number of sectors to read
0141:       0031          ; Max #+1 of secs that may be skipped searching
0142:       0432          ; Verify header field
0143:       0001          ; word count-1 of header field
0144:       010B          ; address of header field
0145:       001C          ; Abort on Not Ready. Write Fault, Overrun
0146:       0003          ; skip to next sector if CRC/Vrfy err on header
0147:       0430          ; read label field
0148:       000C          ; word count of label field
0149:       010D          ; Address of label field in IOCB
014A:       001C          ; Abort on Not Ready. Write Fault, Overrun
014B:       043B          ; Write Data Field
014C:       8100          ; Length of data field (256 words)
014D:       0000    *     ; virtual page # of buffer
014E:       001C          ; Abort on NotReady, WriteFault, Overrun
014F:       0420          ; control word to halt hw after each field
0150:       0426          ; control word to find header sector mark
```

Disk IOCB program for Seek

```
0152:       0000    *     ; Number of cylinders to move (negative)
0151:       8000          ; Insure that the seek
0152:       0422          ; completed from before
0153:       0007          ; clear out the status bits that
0154:       0000          ; are not used in a seek operation
0155:       8000          ; send a step pulse
0156:       0000    *     ; (direction is filled in)
0157:       8000          ; finish sending step pulse
0158:       0000    *     ; (direction is filled in)
0159:       0000          ; increment remaining distance
015A:       0152          ; in IOCB field, and skip if zero
015B:       0002          ; Jump back to the step
015C:       0155          ; loop.
015D:       0006          ; Clean up and finish IOCB
015E:       0000
015F:       0000
0160:       0400          ; quit and write status back
0161:       0103          ; into status word
```

Disk IOCB program for Transfer

```
0162:       0005          ; Load parameters from
0163:       0000    *     ; parameter table
```

```
0164:        8000          ; Wait for any pending seeks
0165:        0422          ; to complete.
0166:        0800          ; transfer run of pages
0167:        0002          ; if there was an error,
0168:        0169          ; finish up anyways (we're done!)
0169:        0006          ; Clean up and finish IOCB
016A:        0000
016B:        0000
016C:        0400          ; quit and write status back
016D:        0103          ; into status word.
```

End of file {Eris}<LispCore>Internal>Doc>DLionDiskDriver.TEdit.

Interlisp-D AR Fields

Interlisp-D software support and development uses an "Action Request" data base system for keeping track of bug reports and requests for features. We take feature requests as serious as bug reports -- don't hesitate to ask. Users are encouraged to submit ARs, either via mail, electronic mail, or (for internal Xerox users) directly using the AREDIT facility within Interlisp-D. The following documents the fields we use in ARs and what they mean:

AR identification

Number: Every AR has a number, which increases by one for each AR submitted. AR numbers are *never* recycled, and ARs are *never* deleted. The AR number is automatically generated.

Source: This is an arbitrary string which should contain the name and electronic address of original customer or internal Xerox users reporting problem. Customers should be identified by "Liason (Customer name)", e.g., Raim.pasa (Bill White @ Teknowledge). This field is used in sending mail back to find out more technical details, or to notify people about the change in status. If multiple people report the same bug, each one should be included in the Source field.

Problem Description

Subject: A terse summary of problem, enough to identify it uniquely. "FOO doesn't work" or "Floppy problem" is not good enough. Think of yourself as a newspaper headline writer: "Attempt to write file when Floppy door open causes awful noise". Implementors may change the Subject field as more details about the true nature of the problem becomes apparent. Feature requests generally start with "Want", e.g., "Want way to make windows triangular rather than square."

Problem Type: What kind of problem report or feature request:

Bug	The system does not work as documented.
Implementation	The system works, but the internal structure is wrong. (Generally submitted by other implementors or looking at the sources.)
Interface	The design of the user interface is wrong. Includes problems in the way in which things display, as well as program callable structures.
Feature Request	Request for a new feature or set of facilities
Documentation	The system works, but the documentation is wrong, unclear, or incomplete.
Performance	The system works, but it is too slow doing the described operation.

Description: This field should contain the complete description of problem or request, including any subsequent discussion. If bug reports come in via electronic mail, put the whole message in this field. Edit relevant info into the beginning of the Description, especially if it summarizes what the problem really is.

Frequency: How reproducable is the problem? Leave blank if irrelevant (e.g., feature request.) Generally only relevant for bug reports. One of:

Everytime	reproducible every time.
Intermittent	doesn't always happen.
Once	saw it happen once.

Impact: How seriously does it affect your ability to get work done, value of Interlisp-D, etc. The names apply to bug reports, but feature requests should be rated along analogous lines.

Fatal	causes system crash, loss of work, etc. requirement for project completion.
Serious	can be worked around but seriously interferes with work, requires substantial reimplementation
Moderate	tolerable, but clearly a problem, responsibility of Interlisp development

Annoying annoying problem, minor request for new feature that "would be nice"
Minor may be some dispute about whether it is even a bug, very minor feature request.

Test Case: This field isn't used for what the name might imply: it should be a list of the files needed to recreate problem. Recipe for reproducing the problem (which is what you might think a Test Case was for) should be in the Description.

Lisp Version: This should be either the release name (Fugue.1, Carol, Harmony) and MAKESYSDATE in which the problem occurs (or the feature doesn't occur.) The Lisp AREDIT package attempts to fill this in; if you submit from a different system that you are running in, please change it. If its a documentation problem, include the date of the documentation.

Machine: What machine does problem occur on (one of 1108, 1132, 1100). Leave blank if *all*.

Disk: What kind of disk is on the machine? (only fill in if relevant to AR.) Constrained to have the known disk types for Machine.

Microcode Version: (automatically generated, delete if known to be irrelevant, e.g., for documentation problems.)

Memory Size: (automatically generated. Delete if known to be irrelevant.)

File Server: If problem deals with communication, what server are you talking to? (This field should really have "Server" on it, rather than File Server.) One of VAX/VMS, VAX/UNIX, 8037, etc.

Server Software Version: As appropriate for the server you're talking to.

System: Subsystem: Category & sub-category of problem type. Subject to change. System includes: Lisp Software, System Software, Text and Graphics, Documentation, IO Architecture. Generally these are filled in by LispSupport, as it corresponds more to our own internal project structure.

Problem disposition

Workaround: If relevant, this field can contain a known procedure to work around problem until it is fixed; generaly a short recipe belongs here.

Status: Status of AR as it moves thru the system:

New All ARs start out as New.
Open Has been looked at by LispSupport; all fields are filled in & has been assigned.
Fixed problem fixed, in LispCore loadup. The In/By: field is set to the next release name. The Assigned-to: field is set.
Closed System with fix in it has been tested & released.
Declined Request for feature officially *never* going to be implemented (e.g., we think its a bad idea). Bug report considered spurious (we don't think it is a bug)
Superseded Another AR includes the problem described in this one. In this case, the Subject of this AR should include the AR# of the one that supercedes it, and the superceding AR should contain the union of information in this one.

Obsolete e.g., module in which problem reported is no longer supported.

Incomplete The information submitted is not enough to take action -- not enough information to identify the bug, or the feature request doesn't spell out in enough detail what is wanted. This is different from Declined in that the request is considered valid, but open for more detail.

In/By: What version of Interlisp-D has/will have this problem fixed? (E.g., Harmony, Intermezzo, Jazz, Fugue.6, etc.)

Disposition: Brief notes explaining changes to status, plus automatically generated description of who changed status when.

Difficulty: Rough estimate filled in by developer on scope of problem.

Easy	< 1 week to fix
Moderate	< 1 month to fix
Hard	< 6 months to fix
Very Hard	> 6 months to fix
Impossible	can’t be fixed
Design	requires a design

Priority: Development’s estimate of whether it will be in the "next" release. (Changes from release to release).

Absolutely	Release will be held if not completed.
Hopefully	Major release goal. Schedule slip admitted, but likely to get completed.
Perhaps	Will get implemented if other revisions in same area are completed.
Unlikely	Unlikely to be included in the next release.

Assigned To: The developer who ’took care’ of this AR, either by fixing it or declining it. Currently only after AR is fixed.

Attn: The developer(s) who should look at this AR. We’re moving away from relying on this.

Submitter: The user name (and registry) of the person who actually did the Submit. Automatically generated by AR submit, not editable.

Date: The date the AR was originally submitted. Automatically generated by AR submit, not editable.

Edit-By: person who edit this AR last. Automatically generated by the Lisp AR edit tool. (The Disposition field contains more info about edits history.)

Edit-Date: date when this AR was last edited. Also automatically generated.

Maintenance Panel Error Code Summary for Xerox 1108 Interlisp-D

There are two types of maintenance panel codes: progress codes and error codes. Progress codes are placed in the Maintenance Panel at various stages of initialization. Error codes are traps which freeze or blink the error number in the maintenance panel. All errors except the 9000-range errors are fatal.

Summary of MP code ranges

0000-0499 boot-time diagnostics
0500-0699 IOP code
0700-0899 Pilot microcode
0900-0999 Pilot
1000-6999 tech-rep diagnostics
7000-8887 Star
8888-8888 MP lamp test
9000-9999 Lisp

Boot-time errors

0096	Insufficient real memory (<1MByte) for lisp
0149	Usually right after power-on. Disk not ready. Safe and effective to 0-boot from this state.

0200-0299 Booting phase 2 (Initial microcode)

0200	normal booting phase 2
0201	CP error in reading from boot device
0202	null Mesa germ installed in physical volume
0203	broken rigid disk boot chain (possibly intermittent)
0204	Illegal IOP port command
0205	CP Trap (CS parity or double-bit memory error)
0206	null diagnostic microcode in physical volume
0207	null Pilot/Mesa emulator microcode in physical volume
0208	null Mesa germ installed in physical volume
0217	Inconsistent Virtual Memory. Requires re-installation or try another partition.

0500-0502 Domino progress codes

0500	StartDomino	Domino has started
0501	InitReadTOD	Domino starting to read the TOD clock
0502	InitReadTODdone	Reading of TOD clock completed (next MP number from Lisp)

0505-0599 Domino error codes

0505	CSParity	CS parity error detected
0506	BurdockCPDisabled	Burdock attempted to use EtherKludge
0507	CPBurdockDisabled	CP attempted to use EtherKludge
0508	IOPBreak	An IOP break with no IOP kernel
0509	IllegalIOPIntr	Illegal IOP interrupt
0510	BadMapEntry	Incorrect vm Map entry in IOP access.
0511	NoCPDmaComplete	CP Dma operation failed to complete
0512	NoCPDmaChannel	CP Dma channel not specified
0513	ReadCPPortDead	CP not responding to Read CPPort
0514	WriteCPPortDead	CP not responding to Write CPPort
0520	StackOverflow	A task’s stack has overflowed
0565	InvToneCmd	Invalid keyboard tone generator comnd
0570	InvProcCmd	Invalid cmd value in Processor CSB
0571	UnImplCmd	Unimplemented cmd in Processor CSB
0572	SetTODError	The Time-Of-Day could not be set
0576	LSEPCtlOVR	LSEP Control CSB overrun
0580	NoValidCommand	Invalid floppy IOCB command
0581	UnImplFloppyCmd	Unimplemented floppy IOCB cmd
0582	InvalidEscapeCmd	Invalid Escape floppy cmd
0583	CommandTrack	Floppy track register is not correct

0584	TrackToBig	Floppy track number is too large
0585	BadDmaChannel	Couldn't program Floppy Dma
0586	NoDmaEndCount1	External Dma End Count not set
0587	NoDmaEndCount2	Internal Dma End Count not set

0900-0999 Pilot codes

0915 Pilot breakpoint

0937 Trying to find out the time and date. Will hang in this state if no time server is responding, and the time has not been set on the machine since power-up.

0981 Trying to discover Ethernet pup host number. Will hang in this state if non-Lisp code tries to perform Pup operations and no Pup ID Server responds.

9000-9299 DLion Interlisp-D microcode error detected

Most of these errors are indicative of some serious problem, probably hardware, and usually fatal (but try ^D if you can't TeleRaid). The main exception is 9004—see description of code 9304.

9001	CSParErr	Control store parity error
9002	StackErr	hardware stack overflow
9003	IBEmptyErr	instruction fetch unit empty error
9004	VirtAddrErr	Attempt to reference virtual address >22 bits
9005	EmuMemErr	double bit memory error or non-existent memory
9009	CAR/CDR	bad pointer
9013	NegPcError	inconsistent PC at FnCall
9014	applyUfn	arg to apply not integer
9016	notFreeTrap	stack allocation error
9024	Page fault in the page fault handler.	
9051	BadUfnTable	
9120	MiscErr Output	opcode no such register
9121	MiscErr	opcode bad 2nd byte
9122	MiscErr Input	opcode no such register
9126	PcNegError	inconsistent PC at Punt

9300-9399 Lisp system code error (call to \MP.ERROR)

These codes generally indicate an error state in Lisp system code that cannot be handled in the break package. Most are "should never happen" cases that indicate a serious error; but some (in particular, 9305 and 9318) may be much less serious. If possible, use TeleRaid to find out more information (press the Undo key to enter the TeleRaid server (cursor changes into "TeleRaid"), and run the TeleRaid user from another machine). Even if you can't TeleRaid from another machine, several of these codes you can convert into a Lisp break if the world is still mostly consistent and the error occurred under user code (rather than, say, the garbage collector): type ^B to the TeleRaid server. Summary of TeleRaid server commands:

^B attempt to enter Break. If error is in a special system context, will change cursor to "CANT", indicating refusal to enter break. Warning: even if the system is willing to try to enter a break, it may fail, leaving your system unrestartable. When in doubt, use ^D.

^D perform Hard Reset—clear stack, flush all non-restartable processes.

^N continue from error. Warning: You should not use this command except for the following errors: 9318 (when you believe it be be continuable, see below); 9915 error when caused by typing the Raid interrupt; 9325; 9326; 9329.

^P display Pup host number (in decimal) in maintenance panel.

9302 Invalid Vmem: attempt to boot an image that is not a valid Lisp sysout, or which is inconsistent from having some, but not all, of its dirty pages written. Can happen if you boot instead of calling `LOGOUT`. Usually caught sooner as code 217.

9303 "No place for IOCB page at startup"—this usually only happens if your machine has insufficient memory.

9304 Obsolete [Map out of bounds].

9305 Invalid address: attempt to use a pointer that does not refer to an existing (allocated) part of virtual memory. Usually means garbage was fetched from somewhere that should have contained a pointer; a common source is code with type checking turned off attempting to fetch a datatype field from an object that is not a datatype, such as `NIL` or a small integer. This error can often be converted to a break with the ^B TeleRaid command if the Lisp image is otherwise in a good state.

9306 Obsolete [Invalid virtual page].

9307 "Unavailable page on real page chain"—inconsistent state in page fault handler.

9308 "Loop in \SELECTREALPAGE"—inconsistent state in page fault handler.

9309 Attempt to allocate already existing page (from call to `\NEWPAGE`).

9310 A 9309 error recursively inside the new page allocator.

9311 "Locked page occupies a file page needed to lock another"—bad state in virtual memory system.

9312 Arg to `CLOCK0` not an integer box.

9313 Fault on resident page: processor took a page fault for a page that appears to be resident.

9314 PageFault on stack: shouldn't happen, as stack is resident.

9316 Obsolete [Attempt to extend vmem beyond 8MB].

9317 "Attempt to write a locked page when not under `\FLUSHVM`"—bad state in virtual memory system.

9318 Error in uninterruptable system code: an error that ordinarily would enter a break (e.g., a type test failure), but in a piece of code that should not be user-interruptable. This is generally a sign that some datum used by system code has been smashed, but this is not always fatal. Should you not be in a position to diagnose the error with TeleRaid, you can type ^N after entering the TeleRaid server; Lisp will proceed from the MP halt and attempt to enter a break anyway, from which (if it succeeds) you might be able to glean more information about the problem. Warning: continuing with ^N can be fatal if the error really was in a place where a break would not succeed.

9319 Stack full: hard stack overflow. A soft stack overflow (Lisp break "`STACK FULL`") occurs when the stack is mostly used up; if you proceed beyond that point without resetting you can completely fill the stack and get this MP code. Press `STOP` to perform a `HARDRESET` to clear the stack, or run TeleRaid to find out who was guilty of overflowing the stack.

9320 Storage is completely full. A continuable Lisp break "`STORAGE FULL`" occurs when the allocation space is nearly full.

9321 Unknown UFN: attempt to execute an unimplemented opcode. This usually means that the processor is trying to execute random memory, or took a wild jump somewhere. Often a microcode bug.

9322 Atoms full: the limit on number of litatoms (2^16) has been reached.

9323 Obsolete [Pnames full].

9324 Stack frame use count overflow: the program has attempted to create more than 200 references to the same stack frame.

9325 Storage nearly full: this is a warning that comes later than the "`STORAGE FULL`" break but before you completely run out (and get a 9320). You can continue from this error with ^N from TeleRaid.

9326 Bad MDS free list: the free list of recycled MDS pages got trashed. You can continue from this error with ^N from TeleRaid.

9327 Bad array block. The array allocator found a bad array block in its free list. Generally means some unsafe code trashed one or more locations in array space.

9328 A variation on 9327.

9329 The garbage collector attempted to reclaim an array block, but the block's header was trashed. You can continue from this error with ^N from TeleRaid, but it is symptomatic of array trashing, and you should save your state as soon as possible and restart in a good sysout.

9330 Reference counting problem: an object marked as having a overflowed reference count (greater than 62) is not found in the overflow table.

9331 Reference counting problem: an object whose reference count just now overflowed was already in the overflow table.

9332 Reference counting problem: an attempt was made to decrement the reference count of an object whose reference count was already zero.

9333 One of a number of consistency checks in the process manager failed.

9334 The process manager needed to build a function frame for some operation, but failed. This normally should never happen, but could conceivably if you are about to completely fill up the stack (and thus would otherwise get a 9319 error).

9335 Occurs at boot time when the sysout you are trying to run uses the full 32MB virtual address space, but you are trying to run it on a machine that can only address 8MB. The function `32MBADDRESSABLE` reports whether a machine has the hardware required to address 32MB.

9336 Somehow control was transferred to the T frame at the top of the world (effectively a `(RETTO T)`, except that `RETTO` turns that case into a `RESET`), thereby evading the process world. This leaves the stack in an unresumable state.

9337 The process that is being scheduled to run next has had its stack released—inconsistent state in the process scheduler.

9393 (Koto Only) See 9341

—Post-Koto Error Codes—

9338-40 Error in locked page logic—not currently used.

9341 Hard disk error in the swapper—the swapper has tried several times to access a page of the virtual memory backing file and failed; page fault cannot proceed.

9342 Disk run table for the virtual memory backing file is malformed.

9400-9899 unassigned

9900-9924 Attempt to call Raid or 1132 Subr.

The only code normally seen in this range is 9915:

9915 Call to `RAID`. Note that if you have the Raid interrupt enabled (typically on ^C), you will get a 9915 error by typing that interrupt character, which you can continue by typing ^N from TeleRaid. Any other occurrence of 9915 generally signifies an error in system code that has not been explicitly assigned a code in the 9300 range.

Subject: Re: Efficiency Hacks?
In-reply-to: LispSupport.pa's message of 4 Jun 84 17:12:29 PDT (Monday)
To: LispSupport.pa
cc: JonL.pa

If Mitch Model's conjectures are true, then we have a a few more good reasons to proceed with the extension to litatom capabilities in Interlisp-D. I have three main points to make, the second of which is rather lengthy, so I'll warn you in advance:
   1) How to find out how many litatoms your sysout has
   2) The importance of swapping in the litatom-hash case
   3) Performance tools: how to find out if you are being bitten by (2).

1) They think that they have about 30000 litatoms -- how do they know? Currently, the value of the variable \AtomFrLst (minus about 1) will be the number of litatoms created in a given sysout. As we are all aware, Interlisp-D has no mechanism for reclamation of "dead" litatoms. If they are approching this limit, then they will be the third *** major *** AI project which recently had to undertake a serious revision of implementation, due solely to the limit on litatoms (i.e., litatoms are the "natural" structure wanted, and the projects converted only after bumping into the hard limit).

2) They say that their subjective impression is that litatom hashing is slow. They could well be right. They should be pointed to ATOMHASH#PROBES, which is in the Lisp.sysout now (but may still be lacking in documentation?).

A good hashing algorithm, such as that in Knuth's book on sorting and searching, will have a small average-number-of-probes and a fairly short "tail", even when the table is around 85% full; as I mentioned in a note to LispCore^ last week, Interlisp-D's litom-hash algorithm produces a fairly small average-number-of-probes but an *** incredibly long "tail" ***. So the problem isn't just that they have a lot of litatoms; rather, it's that they are referencing ones that live on the tail (of course, the problem would be moot if there were enough real memory to hold the full virtual space).

The importance of knowing how many probes it will take to locate an atom can't be overlooked. If found immediately, then the time will be bounded (at worst) by the time to swap in the relevant page of the litatom hasharray, the page of pname-pointer space, and the page of pname-character space; if we take 40ms as an average page-fault time, then MKATOM shouldn't take longer than about 1/10 of a second. [on a Dorado, when no swapping is involved, litatom hashing takes about 110us plus 25us for each probe.] On the other hand, there are 256 pages of litatom hashtable, 256 pages of pname-pointer space, and at least as many of pname-character space; a "worst case" could take over 750 faults, with a time of about 30 seconds **** due entirely to a poorly-performing hash algorithm ****.

That is, we have three-orders-of-magnitude slowdown; and we can't blame "pageing" in general, because a litom-hash should cause at most a couple of page faults -- not a couple hundred faults!

Is such a "worst case" realistic? Will it ever happen in any notable frequency? The answer, unfortunately, is yes. Litatoms created "last" have a higher probability of falling into the long "tail"; thus the ones of "your" application, which you use all the time, will likely be the ones to suffer the "whip of the tail".

Consider some statistics taken from the recent Full.sysout, with my working environment loaded:

```
   2.9   is the expected number of probes, averaged over all atoms
   75%   of all atoms were found in 2 or fewer probes
   95%   of all atoms were found in 12 or fewer probes
```

Those statistics look "pretty good", *** but the remainder 5% were about equally distributed on the interval up to 207 probes ***. In fact, my favorite meta variable, FOO, is the one entry at 207! It took a long time to convince myself, let alone convince others, that the 10 second pause I experienced when typing

    (SETQ FOO 5)

was due to litatom hashing, and not to DWIM, or GC or any of "the usual suspects". This is on a Dorado where, other things being equal, the average page-fault time is more like 25ms [I observed about 49ms for Dolphin and 37ms for DLion with 40MB Quantum disk. But remember, these are "one-shot" averages.]

Compare those 10 seconds with the usual few 10's of milliseconds for this action!

3) There is a difficulty in even noticing that litatom hashing is the problem, because of the interaction with virtual memory problems; one has to have a "nose" for this sort of smelly problem.

I tried the usual metering tools on the forms (MKATOM "NIL"), which takes only one probe, (MKATOM "T") which takes 0 probes (single character litatoms are handled separately), and (MKATOM "FOO"), which takes 207 probes.

-- SPY's output is useless; it merely said that 100% of the time was being spent in MKATOM.

-- DOSTATS, if it were working correctly, would be equally useless. It conveniently filters out the page swap time; only by a "bug" in the filtering did it leak out that 43% of the time went into \WRITEMAP (and as it happens, \WRITEMAP time seems to be less than 25% of time spent in swapping).

-- TIMEALL at least told me about the gross page faulting behaviour

Without suspecting pageing, one might be tempted to interpret the above results as saying that the probe comparison is poorly coded; yet more persistent analysis shows that it costs only about 61us per probe (on a DLion) when faulting isn't involved -- so 207 probes on a DLion should cost at most 14ms, rather than 14+ seconds which does happen to me frequently.

By judicious use of \RELEASEWORKINGSET and a sub-piece of TIMEALL, I found that, in my environment, a "fresh" lookup of T takes 5 faults, NIL takes 8, and FOO takes 323.

The confounding thing is that in a Full.sysout, before I load my applications, there are 20800 litatoms, and the lookup of the worst-case litatom (again, FOO!) costs only 55 probes and 95 page faults. Why should loading a "small" application, with only about 5000 litatoms, blow the whole thing up? The answer must be that the hash algorithm begins to break down at an occupancy level of about 70%

So one can merely recommend that a user snoop around with ATOMHASH#PROBES to "indict" the lookup of some of his favorite atoms.

-- JonL --

LocalFile
1108 Local Hard Disk File System
Intermezzo Release

Stored as [eris]<lisp>intermezzo>doc>localFile.tedit

Introduction

The 1108 hard disk file system is designed to provide Interlisp-D users with a flexible mechanism for storing and accessing files. Like the file systems for the 1100 and 1132, the 1108 file system supports features like random access and version numbers on files. In addition the 1108 local file system supports a hierarchical naming structures for files.

Partitioning the Disk

The hard disk used with an 1108 may be partitioned into up to ten regions called logical volumes. Logical volumes are like directories on the disk device: they may be used to hold Interlisp virtual memories, Interlisp files, Mesa files, or Star files. You can partition the disk with the *Installation Utility* floppy or with *Othello*. Since partitioning the hard disk erases all its contents, you are advised to partition the disk appropriately before storing anything on it; otherwise, you will have to offload all files from the disk, repartition, and then copy the files back to the disk.

Although an Interlisp virtual memory file could coexist on a logical volume with other files, it is generally adviseable to give each virtual memory file a logical volume which it does not share with anything else. Otherwise the resulting fragmentation would adversely affect swapping performance. A logical volume intended to contain an Interlisp virtual memory should be between 16,000 and 64,000 disk pages long.

The Intermezzo file system (unlike its predecessors) allows Interlisp user files to coexist on a logical volume that contains Mesa or Star files. So it is no longer necessary to have a special logical volume given over to Lisp user files, though you still can have one if you like. Note that to store Interlisp files on a logical volume, you must create a Lisp directory on that volume (see below for instructions).

File System Utility Functions

So long as there is a logical volume with a Lisp directory on it, you will have a local disk device called {DSK}. This device can be used from within Interlisp-D just like the {DSK} device on the 1100 and 1132, except that it supports a hierarchical naming structure for files.

If you do not have a logical volume with a Lisp directory on it, Interlisp will emulate the {DSK} device by a coredevice, which is fine except: (a) The coredevice provides limited scratch space for some system programs; (b) when running GREET, Interlisp will fail to find {DSK}INIT.LISP and will have to prompt the user for an init file; and (c) since the coredevice is contained in virtual memory, it (and the files stored on it) can last only as long as you keep your virtual memory image.

To create a Lisp user file directory on a logical volume, call the function

(CREATEDSKDIRECTORY volumeName) [function]

CREATEDSKDIRECTORY affects only the specified volume, and (unlike previous versions of the file system) does not affect Mesa or Star files in the specified volume. CREATEDSKDIRECTORY returns the name of the directory created. Installing an Interlisp directory is something that should have to be done only the first time the logical volume is used. After that, the system will automatically recognize and open access to the logical volumes that have Interlisp directories on them.

Should you ever want to get rid of a Lisp directory (and all the files in it), call the function

(PURGEDSKDIRECTORY volumeName) [function]

PURGEDSKDIRECTORY affects only the Lisp files on the specified volume; it will not tamper with Mesa or Star files on the same volume. An alternative but cruder way to get rid of a Lisp directory is to use *Othello* or the *Installation Utility* to Erase the entire logical volume.

To find out if a particular logical volume already has a Lisp directory on it, call

(LISPDIRECTORYP volumeName) [function]

To find out what logical volumes you have on your local disk, call the function

(VOLUMES) [function]

To find out the total size of a logical volume in disk pages, call

(VOLUMESIZE volumeName) [function]

To find out the number of free pages left on a volume, call

(DISKFREEPAGES volumeName recompute) [function]

And to find out which logical volume contains the virtual memory you are currently running in, call the function

(DISKPARTITION) [function]

Finally, once an Interlisp directory has been installed on a logical volume, any program running in Lisp has access to the Lisp files on the the volume. Access is provided through the usual device independent file interface: CONN (to connect to any directory or subdirectory on the local disk), OPENSTREAM, CLOSEF, DELFILE, GETFILEINFO, SETFILEINFO, BIN, BOUT, LOAD, etc.

File Name Conventions

Each logical volume with a Lisp directory on it serves as a directory of the device {DSK}. Files are referred to as

{DSK}<*logical-volume-name*>*file-name*

Thus the file Init.lisp on the volume LispFiles would be called {DSK}<LISPFILES>INIT.LISP.

In addition, you can indicate subdirectories using the **>** character in file names to delimit subdirectory names. Subdirectories allow users to group files to a finer degree of granularity. Files with subdirectories are written

{DSK}<*logical-volume-name*>*subdir1*>...>*subdirN*>*file-name*

For example, suppose you had a file LRdesign.tedit on the subdirectory ParserGenerator on the subdirectory Compiler on the directory (logical volume) LispFiles of the hard disk device; its name would be written {DSK}<LISPFILES>COMPILER>PARSERGENERATOR>LRDESIGN.TEDIT.

You can default directory names for the 1108 hard disk in an unusual but simple way. That is: *if the file does not have a subdirectory and you leave out the directory (logical volume) name, the directory will default to the next logical volume which has a Lisp directory on it including or after the volume containing the currently-running virtual memory*. Thus if your disk has logical volumes Lisp, Tajo, and LispFiles, and the Lisp volume contains the running virtual memory, and only the LispFiles volume has a Lisp directory on it, then {DSK}INIT.LISP will refer to the file {DSK}<LISPFILES>INIT.LISP. All the utility functions presented above default logical volume names in a similar way, except for those that can't, such as CREATEDSKDIRECTORY. If you want to find out what the default Lisp directory is, call

(DIRECTORYNAME '{DSK})

This defaulting convention is necessitated by several parts of the Interlisp system which create scratch files on the device {DSK} without specifying a directory (logical volume).

Displaying File System State

DSKDISPLAY is a library package which provides a display window for 1108 local file system. The window keeps track of what logical volumes you have on your local disk, which ones have valid Lisp directories on them, and how much space is left on each volume.

The file system display can be in one of three states: ON, OFF, or CLOSED. ON means the display window is updated whenever the file system state changes. (This continuous updating can slow down the file system significantly.) OFF means that the display window is open, but updated only when the user left-buttons it with the mouse. CLOSED means that the display window is closed and never updated. When the DSKDISPLAY package is loaded, the display mode is set to CLOSED.

The functional interface to the file system display is provided by

(DSKDISPLAY newState) [function]

DSKDISPLAY returns the old state of the file system display, and if newState is one of the litatoms ON, OFF, or CLOSED, then the display state will be changed to newState.

The mouse interface to the file system display is as follows: left-buttoning the display window will update it, and middle-buttoning the window will bring up a menu which allows you to change the display state.

Scavenging

Unlike previous releases, Intermezzo provides full disk scavenging service to guard against the unlikely event of file system failure. There are two classes of file system failure: Lisp directory failure, or lower-level (Pilot) failure. Scavenging service for Lisp directories is provided by the library package SCAVENGEDSKDIRECTORY; scavenging for Pilot is provided by either *Othello* or the *Installation Utility*.

Pilot failures manifest themselves as a "HARD DISK ERROR" break within Lisp. To fix such a failure, return to top level, log out of Lisp, get into either *Othello* or the *Installation Utility*, use the Scavenge option on the logical volume in question, and then reboot Lisp.

Lisp directory failures show up as infinite looping or other aberrent behavior while doing a directory search or enumeration. To repair the directory, return to top level, load the package SCAVENGEDSKDIRECTORY, and call

(SCAVENGEDSKDIRECTORY volumeName) [function]

Should you have any doubt which logical volumes to scavenge, scavenge them all. Should you not be sure which scavenger to use on a volume, use them both, Pilot first, then Lisp directory. Neither scavenger will harm an intact volume.

LocalFile Implementor's Guide
Filed as [eris]<lispcore>internal>doc>localFileImpl.tedit
Written by Tayloe Stansbury 18-Feb-85
Modified by Tayloe Stansbury 15-Aug-85 12:27:46

The Dandlion/Dove local file system is implemented in two files: LocalFile and DskDisplay. User-level documentation for the file system can be found in the 1108 User's Guide or on [eris]<lispcore>doc>localfile.tedit.

The Dandelion/Dove local file system has two layers: the Pilot layer and the Lisp streams layer. The Pilot layer emulates a subset of the Pilot file system, as described in the Pilot Programmer's Manual. The Lisp streams layer implements the Lisp streams specification laid out in [eris]<lispcore>internal>doc>streams.tedit.

The Pilot layer is implemented by three modules in the file LocalFile:

1. LFALLOCATIONMAPCOMS, which keeps track of which pages have been allocated and which are free. LFAllocationMap provides the functionality of [idun]<apilot>11.0>pilot>private>volallocmapimpl.mesa, though its implementation is only very loosely based on that file.

2. LFFILEMAPCOMS, which keeps track of the mapping between file ID numbers and runs of disk pages. This mapping is stored in a specialized B-tree. LFFileMap provides the functionality of [idun]<apilot>11.0>pilot>private>volfilemapimpl.mesa, though its implementation was based more on [idun]<apilot100>pilot>private>volfilemapimpl.mesa, and later updated to be compatible with the Mesa 11.0 release of Pilot.

3. LFPILOTFILECOMS, which has a primitive notion of file, as embodied in its datatype FileDescriptor. LFPilotFile handles things like creating, extending, shrinking, and deleting files; reading and writing file pages; labels; and volume root directories (which map file types onto higher level directories -- e.g. Lisp file type -> Lisp directory ID, Mesa file type -> MFile directory ID, etc.). LFPilotFile does not emulate any particular Mesa file, but rather grew up as the gray area between the two layers became more well-defined during the evolution of the Lisp local file system.

The Lisp stream layer is defined by three more modules in the file LocalFile:

1. LFDIRECTORYCOMS, which implements the Lisp directory. The Lisp directory maps symbolic Lisp file names onto Pilot file ID numbers, and handles directory search and directory enumeration.

2. SCAVENGEDSKDIRECTORYCOMS, which implements a scavenger for the Lisp directory. It works by purging the old Lisp directory, creating a new one, using the BTree to figure out what Lisp files there are on the volume, using the leader page of each Lisp file to figure out what its name is, and then inserting an entry in the new directory for each Lisp file. There is no Pilot-level scavenger implemented in Lisp; for that we rely on the Othello Scavenge Logical Volume command.

3. LFCOMS, which implements all other operations of the local disk file device. LocalFile uses Pilot files as backing files for Lisp streams: page 0 of the Pilot file becomes the stream's leader page (containing stuff GETFILEINFO and the scavenger will be interested in), page 1 of the Pilot file becomes page 0 of the stream, etc. Pilot backing files may be longer than the Lisp stream they hold.

In addition, the file DskDisplay provides a window which displays file system status. This file is separate because it relies on the window system and therefore must be loaded considerably later in the loadup process than LocalFile need be.

Some future projects for the file system (apart from fighting off the stream of ARs):

1. Modify the READPAGES and WRITEPAGES methods to transfer contiguous pages all at once, and set the MULTIBUFFERHINT to be T.

2. Rewrite DiskDlion (which implements the Dandelion/Dove disk head) so that disk requests that cross cylinder boundaries are handled in runs rather than a page at a time (for both the Dandelion and Dove). Without this, large disk requests (which happen especially during deleting) can tie up the system for quite a while.

3. Modify DiskDlion to do a process switch while waiting for the disk. Currently it busy waits.

4. Currently too much of the file system is uninterruptable. Unfortunately, the primitive UNINTERRUPTABLY now prevents process switch in addition to preventing keyboard interrupts. What we really need is a construct that will do the latter but not the former. (There is now an AR for such a beast.) Given that, we should remove as many calls to UNINTERRUPTABLY as possible.

5. Currently files are allocated 20 pages at a time, regardless of whether the openfile request came with a size hint. This was originally because allocating long files sometimes took long enough that NS connections got dropped. Once #2 is accomplished, long allocations should be quite a bit faster though; and #s 3 and 4 will make it possible for other processes to sneak in while allocations are going on. Then it would make sense to change the file system to allocate files all at once when a hint is provided.

6. Currently, allocation map buffers and file map buffers are written out only once per stream creation or deletion. Without automatic built-in scavenging, that strikes me as a bit unsafe (although it does speed things up some). It would probably be better to write them out once every allocation or deallocation (and the performance penalty for doing so will not be too great if allocation requests are larger as a result of #5).

7. Longer term: rewrite the directory so that it uses some form of tree search. The linear search currently used gets unacceptably slow for large directories.

Should you have any questions, do not hesitate to contact me. My mail address is Stansbury.pa, and my extension is 4330. Good luck!

Making a Patch for Distribution

(1) Fix the Ar(s) in <Lispcore>Sources>, <Lispcore>Library>, the microcode, or the emulator, whichever is appropriate.

For Library-file patches:

(2) Move the entire LCOM/DFASL file(s) onto <Lispcore>Patches>.

(3) In the Workaround section of each AR you've fixed, note the name of the file(s) you moved onto the patch directory, and the date/time you moved them.

For Lisp system-code patches:

(2) Decide which functions, variables, etc. are needed for the patch file. Determine what pre-requisite patches (if any) need to be applied before this one makes sense.

(3) Make sure you have the latest RELEASETOOLS loaded (the Medley sysout as Cheryl made it doesn't).

(4) Use the "Patch" command to build a patch-file template:

patch *ar1 ar2* ...

where *ar1*, *ar2*, etc. are the AR numbers of teh ARs this patch will fix.

Patch will create the COMS for a file named `AR-`*ar1*`-PATCH`, and will bring them up in an SEdit window for you to edit.

(5) Edit the COMS for the patch file to include all the functions, variables, etc. that are needed.

(6) Connect to {Eris}<Lispcore>Patches>.

(7) MAKEFILE and compile the patch file there.

(8) In the Workaround field for each Ar you fixed in this patch, note the name of the patch file and the date & time you made it.

For microcode patches:

For emulator patches:

Hardware:
- Bigger Disks
- Faster local IFSs
- Color display
- Sell better peripherals
  - Laser writer
  - better file server
  - SCSI
- Labelless Disks
- Streaming Tape

Documentation:
- Updated and online doc
- Improved documents
- Consistent paradigm for writing documents (IM formatting replacement)
- Make system easy to learn

Window system / User interface:
- Interruptable pop-up Menus
- TIP tables
- Remote Windows
- Faster text display
- Imager
- Image Streams
  - clean up code
  - collapse common code
  - add clipping
- Rethink & overhaul Image Objects
- Loopsified window system
- Consistent look & feel (window UI and commands)
  - User interface as consistent as Viewpoint or Macintosh
  - Same keying or mousing gets same op everywhere
- Cheap, Pervasive Text (TTYIN replacement)
- Symbol ESC completion on type-in

Programming Environment:
- Same machine low-level debugger
- Source level debugger
- Good stepper
- NS based Teleraid
- Teleraid and CL stack ops
  - revert
  - return from
- Teleraid Inspect
- Interactive Interface to File manager
- Better File Browser
- Definition groups
- Extensible SEdit
- Programmers Interface to SEdit
- More Common Lisp integration

Higher level language features:
- Unification
- Single, Common prettyprinter
- Fast prettyprinter
- Path name cleanup
  - use pathnames all the way down
  - get SAME object always for same file
- Single Compiler
- Decouple DWIM - CLISP
- Fast sequence functions
- Ropes

System building and installation:
- Non-Dorado loadups
- FASL in the init
- Packages in the init
- Common Lisp primacy, eg in the init
- State saving smaller than sysout-sized
- Revamp software Installation process
  - better user interface
  - written in Lisp, not Mesa
  - FAST
  - include error Recovery

Language kernel:
- Big Reference counts w/microcode support
- Full GC (circular objects)
- Get arith to IEEE compliance
- GCable Symbols
- Unboxed floating point compler
- Finalization of GCed objects
- Smaller base sysout
- More Common Lisp microcode, eg EVAL
- Common Loops IN the system

Operating system:
- Pre-emptive scheduler
- Non-consing Synchronizers (monitorlocks)

Device drivers:
- SNA networking
- Floppy speed
- Improve local disk speed
- Real Subdirectories
- Support File Cacher
- Reliable TCP & RS232
- Run in XNS only world
- Run in Stand Alone World
- Sun NFS support
- XNS over phone lines

Applications:
- Video Image Manipulation
- Bitmap Editor as good as MacPaint
- Mail support for
  - X.400
  - SMTP
- Release Lafite
- Lisp based File Server
- 3270 terminal emulator
- Good 3rd party software development path
- Database
- Spreadsheet
- AR category clean up
- Programming system Management Tools
- Adobe Tools
- Notecards
- TEdit
  - Footnotes
  - Index
  - toc
  - egns
  - styles
  - LPT output
  - change masks
  - better than Tex

        WYSIWG Page mode
        DES/DIS/DIF/VP convert
    Easier Landscape Printing
    YACC and such
    C, Fortran, Pascal, Cobol

Work environment:
    Work at Home
    Better design process

NS Character Set Issues:

Greg Nuyens. Aug 16,1984

This note describes a proposal for incorporating the NS character set standard into Interlisp-D. This proposal will encompass the following areas:

--implementation of a "long" character data type

It specifically omits the following:

--external file representations. **

--the reworks necessary to fonts, etc. to attain DIG, though this proposal is made with these issues in mind.

Definitions

The following terms are defined for use in the remainder of the document.

character: an instance of the "long" character data type.

byte: an 8 bit unsigned integer corresponding to the current Interlisp-D character.

character set: the (conceptual) array of 256 characters into which all the representable distinct characters have been separated. An example is character set 0 which contains most but not all of the characters currently used in Interlisp. For instance an umlaut is not included in character set 0.

font: A style in which characters are rendered. A font includes (conceptually) an image for every representable character (2^16 - 256). A font is a member of the cross product of Family, Weight, Slope, Face, Nominal Size, Rotation, and Expansion . An example is TimesRoman 12 Bold Italic Regular Compressed with 90 degree rotation. To contrast character set and font, note that only one character set (0 *?*) contains the greek letter alpha, but every font contains a rendering for that character. In practice few (no?) fonts will contain all the images of all the characters.

Overview

Before discussing how these functions will change, an overview of the scheme for the internal functions:

Unknown IMAGEOBJ type
GETFN: SKIO.GETFN

There will be two types of character data, the previously described long and short characters. However, any single string will consist of only one type of character. Thus strings as currently represented, (effectively byte arrays) will continue to exist. However, there will also be arrays of long characters. Any functions which receive arguments of mixed type (e.g. RPLCHAR given an string of short chars and a long char to replace with) will produce results composed of long chars only.

Thus, the current implementation of substrings (as "tails" of strings) will not suffice, since the original string may be coerced upward to a long char string by an operation occurring after a substring has been returned. The plan is to use forwarding pointers in string space and store substrings as <header,offset,length> triplets. Thus when a short char string is coerced upward into a long char string, the original header will have a pointer to the new location. Any substrings returned previous to the coercion will still be valid, since they reference the (now indirect) original header. The offsets will always be scaled by a bytes-per-character value implicit in the type of string (1 for short strings, 2 for long strings).

This scheme guarantees the advertised property of substrings that they are indeed shared tails. That is, a destructive change to a substring will affect the string. This will be true regardless of any coercion of the string that occurs.

Also recognize that any coercions performed on substrings will change the representation of the whole string.

Changes to Interlisp Functions

In previous discussions the two following lists of functions affected by this proposal were identified:

First the functions which deal with the internal representation of character data:

RPLSTRING(X N Y)

this function must change so that if either argument is "long", both are coerced to long. For the character argument, this is simply to add the default character set. For the string however, it will be necessary to copy the string coercing each character by appending the default character set.

RPLCHARCODE(X N CHARCODE)

must now take long or short charcodes, handling them as will RPLSTRING.

GNC{CODE}(X)

will always return an atom representing a long character (Does this mean having all the single character atoms? NO--they can be MKATOMed on the fly.) {or for GNCCODE, a long charcode}

GLC{CODE}(X)

as above

NTHCHAR{CODE}(X N FLG RDTBL)

will always return an atom representing a long character.

NCHARS(X FLG RDTBL)

Returns number of characters independent of representation. When FLG is T the prin-2 length (as yet upspecified for long chars) will be used.

STRPOS{L}(PAT STRING START SKIP ANCHOR TAIL) (and MAKEBITTABLE)

If either PAT or STRING is constructed of long chars, then the comparison will take place as though both were long. However, no destructive changes will be made.

CHARACTER(N)

Always produces an atom whose printname is the long character whose representation is N.

CHCON(X FLG RDTBL)

Still produces a list of charcodes. These may be short or long charcodes

SUBSTRING(X N M OLDPTR)

Performs the substring operation, guaranteeing the EQ invariant for substrings. The internal rep'n will be the header together with the offset and length, so that if upward conversion later happens, shared tails remain.

ALLOCSTRING(N INITCHAR OLD)

As before except, that if INITCHAR > 255 then the resulting string will be a long string. (Coercing the OLD string's char array as needed.)

MKSTRING(X FLG RDTBL)
As before, except the PName may be long.
CONCAT{LIST}(U ...) {L*}
If any of the arguments are long strings, the result is a long string.
STREQUAL
This will be true when the characters are the same, regardless of the representation.

and the following functions which must know the external representation of character data

FILEPOS
SETFILEPTR
BIN
READC
\INCHAR
\OUTCHAR
COPYBYTES
COPYCHARS

Efficiency Concerns

the common case of bytes stays fast (though some penalty)

readc is already coercing bytes upward into smallp's why not into long chars?

will it be necessary to have all 2^16 - 256 unit length pname atoms exist. (what does readc currently return)?

The intention is that the common case of short char arguments will retain their current efficiency.

Notes from opcode review meeting (25 June 1986)
Compiled by Jan Pedersen

For Immediate Action:

The following opcodes are not emitted, have no microcode, and can be immediately recycled: (Someone should volunteer to do this and announce to xclispcore)

Opcode#(octal)	Name	
007	CDDR	;; unwind
036	PUTHASH	;; findkey
041	BOUT	;; still seems to be emitted
043	LIST1	;; restlist
044	DOCOLLECT	;; there is apparently a holdout that emits this.
045	ENDCOLLECT	;; ditto.
177	AUDIO	;; retcall
374	RESERVED for Dolphin	

The following opcodes are emitted, but have no microcode -- and may be recycled after recompilation of all sources. (Someone should volunteer to remove all optimizers -- Macros, Dopvals, Dopcodes, etc. -- associated with these opcodes and announce to xclispcore)

Opcode#(octal)	Name
033	GETPROP (?)
035	GETHASH
050	ELT
051	NTHCHC
052	SETA
053	RPLCHARCODE
055	EVALV
160	ATOMNUMBER
313	GETBASEFIXP
314	PUTBASEFIXP

Recycling these opcodes demands recompilation, but this might be an additional reason for declaring .DCOM files not readable in Lute.

For Discussion and Design:

Uncoordinated changes:
(that is, changes which do not alternate the meaning of existing constructs)

(Order of tasks is random -- not priority order)

Approx. time	Task
Day	Add opcode for Read-Char (like NTHCHC, BIN) convert (byte/word) to (Characterp/Smallp)
Day	Add opcode for = Compare two numbers for equality -- does coercions
Day	Add opcode for ASH (arithmetic shift) Arithmeticly shift integer X bits to left

	if X pos. or X bits to right if X neg.
Day	Port LISTGET to 1186 8K microcode
Day	CL versions for ASSOC, FMEMB, EQUAL, (MEMBER) Differ from IL versions in that EQL is used rather than EQ --- microcode may only do EQ test on Symbols and punt otherwise
Day	Debug EQL, ARG0 -- insure correct and efficient algorithm
Week	Complete port of unboxed scalar F.P. opcodes to 1186 (includes basic arith., and comparison)
Week	Microcode more cases for RECLAIMCELL RECLAIMCELL currently only reclaims Cons's -- should be extended to FIXP and Floatp boxes.
Month	Microcode additional CL arith. operators (Truncate and friends) Spec. here not yet clear -- will appear after benchmarking
?	CL:EVAL (like EVAL but for commonlisp)
?	REF
Month	Finish array microcode

Coordinated changes:
(that is, changes which require changes both to lisp and microcode)

(Order of tasks is random -- not priority order)

Approx. time	Task
Week	CREATECELL countdown Modify GC so reclaim occurs after countdown to zero from some settable start - Important for tighter control over GC -
Month	GC hash algorithm changes Address issues of GC hash table overflow - Important for Cons'y benchmarks -
Week	Subtyping in TYPEP Make TYPEP more useful May render TYPEMASK.N redundant
Week	Replace DTEST and TYPECHECK to be more TYPEP like (two type bytes rather than one?)
Month +	FN call changes for &optional, &key, &rest, closures and arg # checking, field descriptors, multiple values, descriminators (Larry, Bill and Pavel have volunteered to work on this)

Interlisp-D Opcodes

```
Written by: Masinter, van Melle, Sybalsky
Stored on [Eris]<LispCore>Internal>Doc>Opcodes.TEdit
RamVersion:        26,24 (recently incremented)
LispVersion:            113400
```

UFNs—Handling undefined op-codes

When the microcode (or C emulator) doesn't handle an opcode, it "punts" to the UFN for that opcode: a Lisp function that does what the opcode should do.

To find out what function to call, the microcode looks at a 256-cell block of storage called the "UFN table" (pointed to in Lisp by \UFNTable). The UFN table contains, for each opcode

(FNINDEX WORD) Atom number (really "definition index") of the function to be called.
(NEXTRA BYTE) # of extra bytes to be pushed as argument to the UFN (either 0, 1, or 2).
(NARGS BYTE) # of arguments to call the UFN function with.

The Op-code descriptions

In multibyte opcodes (len-1>0), alpha is byte 1, beta is byte 2, and gamma is byte 3.

TOS refers to the argument on the top of the stack; TOS-1 is arg one back, etc.

@[x] is the contents of the word pointed to by x.

#	name	len-1	stk level effect	UFN table entry
0	-X-			

used only to denote end of function, never executed.

#	name	len-1	stk level effect	UFN table entry
1	CAR	0	0	\CAR.UFN

If arg not LISTP
  If NIL, return NIL
  else call UFN

If cdr code=0, follow indirect pointer. Take value of car field & return it. (Cons cells are 32 bits: first 8 are cdrcode, rest are "carfield")

[required by diagnostics (except car[NIL]); implemented in all ucodes]

#	name	len-1	stk level effect	UFN table entry
2	CDR	0	0	\CDR.UFN

```
If arg not LISTP
      if NIL, return NIL
      else call UFN
```

if cdrcode=0, follow indirect pointer
if cdrcode=200q, return NIL
elseif cdr code gt 200Q, CDR is on same page as cell,
in cell page+2*(cdrcode-200Q)
else CDR is contained in cell at PAGE+2*(cdrcode).
(Cons cells are 32 bits: first 8 are cdrcode, rest are "carfield")

[required by diagnostics (except cdr[NIL]); implemented in all ucodes]

#	name	len-1	stk level effect	UFN table entry
3	LISTP	0	0	LISTP

Return arg if LISTP (NTYPX=LISTPType), else NIL

[required by diagnostics; implemented in all ucodes]

#	name	len-1	stk level effect	UFN table entry
4	NTYPX	0	0	NTYPX

Return type number of arg (right half of word at `MDSTypeTable + [tos rsh 9]`)

[required by diagnostics; implemented in all ucodes]

#	name	len-1	stk level effect	UFN table entry
5	TYPEP	1	0	\TYPEP.UFN

return arg if type=alpha byte, else NIL

[required by diagnostics; implemented in all ucodes; similar to LISTP]

#	name	len-1	stk level effect	UFN table entry
6	DTEST	2	0	\DTEST.UFN

return arg if typename=(alpha,beta), else call UFN or atom number 372 (`\DTESTFAIL`) with tos and (alpha,beta). (typename is word 0 of type's `DTD`; `DTD` is `DTDBase+(type# lsh 4)`)

[required by diagnostics; implemented in all ucodes]

#	name	len-1	stk level effect	UFN table entry
{ 7	CDDR	0	0	CDDR

TAKE CDR Twice [not currently used or implemented or emitted.]}

REPLACED BY :

7	UNWIND	?	?	\UNWIND.UFN

(N is the alpha byte, KEEP is the beta byte) Unwinds the dynamic stack of the current frame to absolute stack depth N, performing any unbinding indicated by bind marks found along the way. If KEEP is 0, the original top of stack is discarded, otherwise it is pushed after unwinding everything else. This opcode is essentially the same as UNBIND or DUNBIND, except that you stop when the stack depth is N, rather than stopping as soon as you have processed the first bind mark.

The stack depth N is measured in cells (doublewords) starting at the base of the pvar region. N=0 means the stack is utterly empty (including the pvar region; i.e., the end of stack pointer (pointer to next stack block) would be the same as PV). Of course, N=0 cannot be used at all in the present architecture, since there is always at least a quadword pad between the frame header and the start of the dynamic stack. If we get rid of that quadword, then N=0 could have meaning in a frame that had an empty pvar region, though that is not true of any closure target, the current sole user of this opcode.

Note that taking the stack depth as alpha byte means this opcode cannot unwind to any deeper than depth 255. For sake of reference, the largest pvar region in Full.sysout is for the function \CURVE, whose pvar region is 92 cells long (59 locals and 32 fvars), which means it could still achieve a dynamic depth of an additional 173 cells before UNWIND would care (it actually never exceeds a depth of 30).

```
{let SP be the stack pointer; i.e., TOS = @SP}
TOP _ loc[pvar0]-2 + 2*N
if KEEP neq 0
   then TEMP _ TOS
until (SP _ SP - 2) = TOP
  do if @SP is bind mark
        then perform its unbinding
if KEEP neq 0
   then push TEMP
```

#	name	len-1	stk level effect	UFN table entry
10	FN0	2	1	
11	FN1	2	0	
12	FN2	2	-1	
13	FN3	2	-2	
14	FN4	2	-3	

call fn (alpha,beta) with N args [required]

#	name	len-1	stk level effect	UFN table entry
15	FNX	3	FNX	

call fn (beta,gamma) with alpha args [required]

#	name	len-1	stk level effect	UFN table entry
16	APPLYFN	0	-1	

call fn (tos) with (tos-1) args after popping tos & tos-1 [required. Right now, it goes to \INTERPRETER if TOS isn't a litatom. May add requirement that will work with code blocks.]

#	name	len-1	stk level effect	UFN table entry
17	CHECKAPPLY*	0	0	\CHECKAPPLY*

If TOS is a literal atom whose definition cell has CCODEP on and ARGTYPE=0 or 2, return it, otherwise call UFN. Note that CHECKAPPLY* is always immediately followed by an APPLYFn. If it would save some time, you might be able to immediately jump to the APPLYFN code. (note: definition cell: bit 0 is CCODEP, bit 1 is "fast" {this fn has empty nametable}, bits 2-3 are ARGTYPE)

[not required; implemented on Dorado]

#	name	len-1	stk level effect	UFN table entry
20	RETURN	0	0	\HARDRETURN

```
do return except when:
      slow bit in returner is on
       and
```

```
returnee usecount not 0
 or
returners BF usecount is not 0
 or
returnee not immediately followed by
   a free block
     or
   the basic frame of the returner.
```

In any of those conditions, call UFN or context switch to hardreturn context (which?). [required]

#	name	len-1	stk level effect	UFN table entry
21	BIND	2		

push binding mark, bind variables, popping values of stack. [required]

alpha byte is [#NILS <<4 + #BINDS].

beta byte is [FirstPVAR], which is 1-origin (i.e., 0 is PVAR1?? it looks like --JDS)

BIND takes #BINDS values off the top of stack and binds FirstPVAR and successive PVARs to those values. It then sets the #NILS PVARs beyond that to NIL.

Finally, a “binding mark” is pushed on the top of the stack:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|       ~(#NILS + #BINDS)       |         FirstPVAR << 1        |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

Binding marks are identified on the stack because they’re negative: The high bit is guaranteed to be 1.

#	name	len-1	stk level effect	UFN table entry
22	UNBIND	0		

remember tos, pop until binding mark, unbind variables, push old tos [required]

#	name	len-1	stk level effect	UFN table entry
23	DUNBIND	0	(DUNBIND)	

pop until binding mark, unbind variables [required]

#	name	len-1	stk level effect	UFN table entry
24	RPLPTR.N	1	-1	\RPLPTR.UFN

deleteref value at @(tos-1)+alpha.
addref (tos)
store (TOS) at @(tos-1)+alpha [leave high byte of destination intact]
pop (return (TOS-1)).
If reference count failure, call GCTABLESCAN (atom ????) on punt [not required; in Dorado, 12K]

#	name	len-1	stk level effect	UFN table entry
25	GCREF	1	0	\HTFIND

perform ref count operation on TOS according to alpha byte:
- 0 - addref (add 1 to reference count)
- 1 - delref (subtract 1 from reference count)
- 2 - stkref (turn on "stack reference" bit)

If DELREF causes new refcnt to go to 0 & stk bit off, return arg, else always return NIL. On reference count failure, call UFN (no GCTABLESCAN). [not required; in D0, Dorado]

#	name	len-1	stk level effect	UFN table entry

26	ASSOC	0	-1	ASSOC

if TOS=NIL, return NIL.
if TOS not LISTP, call UFN
if (CAR TOS) not LISTP, call UFN
if TOS-1 = (CAAR TOS), return (CAR TOS)
set TOS_(CDR TOS), reiterate, checking for interrupts

[not required, in 12K]

#	name	len-1	stk level effect	UFN table entry
27	GVAR_	2	0	\SETGLOBALVAL.UFN

Do RPLPTR on VALSPACE+2*(alpha,beta) of TOS [not required; in Dorado, D0. May want to change to UFN if high bit of val cell is on]

#	name	len-1	stk level effect	UFN table entry
30	RPLACA	0	-1	RPLACA

if TOS-1 not LISTP, call UFN
Fetch @[TOS-1].
if cdrcode=0, follow indirect
Do RPLPTR with TOS
pop (return (TOS-1)).

[not required; in Dorado, 12K]

#	name	len-1	stk level effect	UFN table entry
31	RPLACD	0	-1	RPLACD

if tos-1 not listp, call ufn
fetch @ tos-1
if cdrcode=0, follow indirect
if cdrcode<200Q
    rplptr cell+2*cdrcode with tos
elseif TOS is NIL
    if CDRCODE#200, deleteref cell+2*(cdrcode-200)
    change cdrcode to 200
elseif TOS is on same page as cell
    addref TOS
    if cdrcode#200, deleteref cell+2*(cdrcode-200)
    change cdrcode to 200+(cell# of TOS)
else (can call UFN on this case)

(this punts on cases where RPLACD must allocate space) [not required; in Dorado, 12K]

#	name	len-1	stk level effect	UFN table entry
32	CONS	0	-1	CONS

Cons pages start with two word header:

```
word 0: [cnt, nxtcell]   (two 8-bit fields: count of available cells
                              on this page, and word# of next free cell
                              on this page)
word 1: nextpage         (page# of next cons page)
```

DTDs (data type descriptors) have (ucode relevant fields in caps)

```
word 0:     NAME
word 1:     SIZE
words 2,3:  FREE
words 4,5:  descrs
words 6,7:  tyspecs
words 10,11:     POINTERS
```

```
      words 12,13:      oldcnt
      word 14:    COUNTER
      word 15:    NEXTPAGE
\CDR.NIL= 200q
```

LISTPDTD is the DTD for type LISTP, i.e., at DTDbase + (LLSH 5 4)

`Subroutine MAKECONSCELL[page]` (given page, return new cell from it):

```
      new cell is at page + page:nxtcell
      new CNT is old CNT - 1; punt if CNT was zero
      new NXTCELL is new cell's cdr code
```

```
Subroutine NEXTCONSPAGE:
      if LISTPDTD:NEXTPAGE # 0 then return it, else punt
      (lisp code scans for page with cnt>1)
```

`CONS(X Y)` // note: this may not be right. Check sources for truth

```
If Y is NIL:
      get NEXTCONSPAGE
      MAKECONSCELL on it
      store new cell with \CDR.NIL in cdrcode (hi byte)
      X in rest of cell

Elseif Y is a listp and the CNT in Y's page > 0, then
      MAKECONSCELL[Y's page]
      store X as CAR, CDR code = ([(LOLOC Y) and 377q] rsh 1) + 200q
Else:
      get NEXTCONSPAGE
      MAKECONSCELL on it
      store Y in new cell (hi byte 0)
      (remember this as Z)

      MAKECONSCELL on same page
      store X in new cell, with hi byte= [(LOLOC Z) and 377q] rsh 1

ADDREF X
ADDREF Y
increment LISTPDTD:COUNTER
DELREF result

[not required, in Dorado, 12K]
```

#	name	len-1	stk level effect	UFN table entry
33	CMLASSOC	0	-1	CL::%%SIMPLE-ASSOC

Takes to two arguments off the stack and returns the of the simplest case of cl:assoc. Equivalent to ASSOC opcode, except punts if the key argument is not an immediate datum. [not required, not implemented on 4K, Dorado]

#	name	len-1	stk level effect	UFN table entry
34	FMEMB	0	-1	FMEMB

if TOS=NIL, return NIL
if TOS is not LISTP, call UFN
if (CAR TOS)=TOS-1, return TOS
else TOS_(CDR TOS), do jump to . [i.e., iterate]
Be sure to allow interrupts.

[not required; in 12K]

#	name	len-1	stk level effect	UFN table entry
35	CMLMEMBER	0	-1	CL::%%SIMPLE-MEMBER

Takes to two arguments off the stack and returns the of the simplest case of cl:member. Equivalent to FMEMB opcode, except punts if the key argument is not an immediate datum. [not required, not implemented on 4K, Dorado]

#	name	len-1	stk level effect	UFN table entry
{ 36	PUTHASH	0	-2	PUTHASH

[not required, not implemented]}

REPLACED BY :

36 FINDKEY

```
alpha = arg#
tos = key
for z from arg# to numargs - 1 by 2
   if arg(z) = key then return(z + 1)
return(NIL)
```

#	name	len-1	stk level effect	UFN table entry
37	CREATECELL	0	0	CREATECELL

Create a new cell of type TOS (a smallposp):
DTD _ DTDSpace + (type lshift 4)
NewCell _ DTD:FREE (2 words)
DTD:FREE _ @(NewCell) (2 words)
 if DTD:FREE is now NIL, signal a gc punt at end of opcode
increment DTD:COUNTER, signal a gc punt if counter goes negative
Zero out DTD:SIZE words starting at NewCell (always an even number)
Deleteref NewCell
TOS _ NewCell

[not required; in Dorado, D0]

#	name	len-1	stk level effect	UFN table entry
40	BIN	0	0	\BIN

If TOS is not of type STREAM (13q) then PUNT

Format of stream is (only some fields are used by microcode):

```
word 0: COFFSET             ; a byte offset from BUFFER
word 1: CBUFSIZE            ; size of input buffer in bytes
word 2&3: flags [byte]  = READABLE (bit 0), WRITABLE (bit 1),
                          EXTENDABLE (bit 2), DIRTY (bit 3),
                          PEEKEDCHARP (bit 4), ACCESSBITS (bit 5-7)
      BUFFER [24 bits]    ; pointer to data
word 4: BYTESIZE
        CHARSET           ; 8 bits each
word 5: PEEKEDCHAR        ; valid when PEEKEDCHARP true
word 6: CHARPOSITION
word 7: CBUFMAXSIZE       ; maximum size of output buffer
```

If COFFSET >= CBUFSIZE then PUNT [buffer overflow]
If READABLE is off then PUNT
Fetch and remember the byte at BUFFER + COFFSET[byte offset]
    Note that this address is guaranteed to be valid at this point,
    but it could pagefault.
Update the stream:
    store COFFSET _ COFFSET + 1
Return the remembered byte as a small positive number.

[not required; in Dorado, 12K]

#	name	len-1	stk level effect	UFN table entry
41	BOUT	0	-1	\BOUT

If TOS-1 is not of type STREAM (13q) then PUNT. (see format under BIN)
If TOS is not a small positive number (< 400Q) then PUNT.
if WRITABLE is off then PUNT

if BUFFER is NIL then PUNT
if COFFSET >= CBUFMAXSIZE then PUNT
deposit byte from TOS at BUFFER + CCOFF[byte offset]
Update the stream:
    store COFFSET _ COFFSET + 1
    set DIRTY flag to 1 [if it isn't already]
return the smallposp one (1)
[not required; not implemented; not even generated by compilers (3/13/89)]

#	name	len-1	stk level effect	UFN table entry
42	PROLOGOPDISP	0	0	none

Implements the Prolog Opcode Dispatch. Uses the Prolog registers PC, N, USQbase, and uSQtablebase.
It takes one arg (DEST). In pseudo-RTL:

```
      if smallp(DEST) then PC _ PC + DEST
            else PC _ DEST
      N _ logand(PC^ 00FF'x)
      opcode _ lrsh(PC^ 8)
      if uLMBase(opcode) = 1 then
            { LispPC _ USQbase + uSQtablebase(opcode)
              return to Lisp }
            else
            { PC _ PC + 1
              (run microcode version) }
```

#	name	len-1	stk level effect	UFN table entry
{43	LIST1	0	0	CONS

(perform (CONS TOS NIL)] not required, not implemented}

REPLACED BY :

43 RESTLIST

```
    alpha = skip  --  number of args to skip
    tos = last  --  last arg#
    tos-1 = tail
    IF tail = NIL THEN
        page _ NEXTCONSPAGE
        GOTO make
    ELSE
        AddRef tail
        page _ CONSPAGE[tail]
        GOTO make
    make:
        get [cnt,,next] from page
    make1:
        tail _ CONSCELL (CAR = IVar(last), CDR = tail)
        AddRef IVar(last)
        IF skip = last THEN GOTO fin
        last _ last - 1
        GOTO make1
    noroomonconspage:
    fin:
        store updated [cnt,,next]
        update ListpDTD:COUNTER
        DelRef tail
        IF noroomonconspage THEN UFN
        ELSEIF ListpDTD:COUNTER overflow then GCPUNT
        ELSEIF overflow entries then GCHANDLEOVERFLOW
        ELSE NEXTOPCODE
```

#	name	len-1	stk level effect	UFN table entry

44	MISCN	2	1 + (-n)	\MISCN.UFN

Miscellaneous opcode for opcodes needing n args from the stack. The alpha byte contains the sub-opcode number and the beta byte contains the number of arguments on the stack. This opcode was added specifically for bytecode emulated implementations, where the opcodes could be written in C. This opcode provides the same functionality of the SUBRCALL opcode, except it has the added flexability of having the opcodes UFN (on both Suns & D-Machines). The UFN vectoring routine is written to adjust the stack according to the number of arguments stated in the beta byte, and there is a UFN handler for each sub-opcode. The opcode is generated using the (MISCN NAME &REST ARGS) macro & optimizer defined in LLSUBRS. The NAME parameter must be registered in \MISCN-TABLE-LIST list, which is of the form (name index ufn-name). The \INIT-MISCN-TABLE function initializes the MISCN's sub-opcode UFN vector.

The predefined MISCN sub-opcodes are as follows:

index	name	function
0	USER-SUBR	This is for the user-supplied subr C coded subrs. It contains its own sub-opcode division based on the 1st argument on the stack. Like MISCN, USER-SUBR requires that the user-subrs be registered with the variable \USER-SUBR-LIST (name index ufn) by calling the \INIT-USER-SUBR-TABLE function. Thus user-defined subrs can each have thier own ufn handler which will be indexed through the MISCN & USER-SUBR UFN mechanism. This opcode can be generated using the (USER-SUBR NAME &REST ARGS) macro found in LLSUBRS.
1	CL:VALUES	Return multiple values
2	CL:SXHASH	Common Lisp hash-bits function for EQUAL hash-tables
3	CL:EQLHASHBITSFN	[Not currently implemented]
4	STRINGHASHBITS	IL hash-bits function for STREQUAL harrayp's
5	STRING-EQUAL-HASHBITS	IL hash-bits function for String-EQUAL harrayp's
6	CL:VALUES-LIST	Return a list of multiple values.

To reserve new MISCN & USER-SUBR entries, you should set the global values for \MISCN-TABLE-LIST and \USER-SUBR-LIST in the LLSUBRS file & re-write the file to insure that you will have unique numbers. The funcion WRITECALLSUBRS whould also be called to generate a new subrs.h file, which contains the C constant definitions for the proper indexes in the C code.

The args to the MISCN UFN routines consist of (INDEX ARG-COUNT ARG-PTR), where INDEX is your sub-opcode number, ARG-COUNT is the number of args to be found on the stack, and ARG-PTR is a pointer to the 1st arg found on the stack. The rest of the args can be found by using (\ADDBASE ARG-PTR (LLSH n 1)) for the n-1th arg.

USER-SUBR UFNs have similar args of (USER-SUBR-INDEX ARG-COUNT ARG-PTR), where USER-SUBR-INDEX is the user-subr sub-opcode index, and ARG-COUNT & ARG-PTR are the same as in MISCN UFNs.

CAUTION: Since the stack affect is variable, thus not known to the compiler, the optimizer may do something funny to the stack args around your call. You should check the emitted code to be sure that things compiled correctly. Putting your calls in small functions will help.

#	name	len-1	stk level effect	UFN table entry
45	<unused>	0	-1	(was ENDCOLLECT)

[not required; not implemented, will be eliminated]

#	name	len-1	stk level effect	UFN table entry
46	RPLCONS	0	-1	\RPLCONS

takes two args (LST ITEM):
check (LISTP LST)
LST's pages CNT field # 0 (see CONS above),
LST's cdrcode = 200q.
call UFN if any of these are not true
MAKECONSCELL on LST's page

store ITEM as in cell, with cdr code = 200q (\CDR.NIL)
store as LST's new cdrcode (((LOLOC newcell) and 377) rsh 1) + 200q.
ADDREF item
increment LISTPDTD:COUNTER
return new cell
[not required; in 12K]

#	name	len-1	stk level effect	UFN table entry
50	ELT	0	-1	ELT

(ELT array index)

Check if TOS-1 is type ARRAYP, call UFN if not
Check if TOS is smallpos, call UFN if not

Array descriptor:
word 0,1: Flags(8),,base(24)
Flags = Orig(1), unused(1), Readonly(1), unused(1), type(4)
word 2: Length
word 3: Offset

Compute index = (TOS) - Orig
if index < 0 or index >= length, call UFN.
index _ index + Offset
dispatch on type (note that index*2 may overflow):
[0] (byte) return (GETBASEBYTE base index)
[1] (smallpos) return (GETBASE base index)
[2] (fixp) return 32 bits at base+index*2 as a fixp (possibly smallp)
[3] (hash) return (GETBASEPTR base index*2)
[4] (code) same as byte
[5] (bitmap) same as smallpos
[6] (pointer) return (GETBASEPTR base index)
[7] (float) return 32 bits at base+index*2 as a floatp
[11.] (double-pointer) same as hash
[12.] (mixed) same as hash
[not required; not implemented yet]

#	name	len-1	stk level effect	UFN table entry
51	NTHCHC	0	-1	NTHCHARCODE

Same as ELT, except type of TOS-1 is STRINGP, the type of the array is always 0, and (optionally) return NIL instead of calling UFN when index is out of range. [not required; not implemented]

#	name	len-1	stk level effect	UFN table entry
52	SETA	0	-2	SETA

(SETA array index value)

Check array and compute index as with ELT.
If ReadOnly is true, call UFN.
In all cases, leave value on stack on exit.
Dispatch on type:
[0] (byte) perform (PUTBASEBYTE base index value)
[1] (smallpos) perform (PUTBASE base index value)
[2] (fixp) unbox integer value, deposit 32 bits at base+index*2
[3] (hash) perform (RPLPTR base+index*4 value)
[4] (code) same as byte
[5] (bitmap) same as smallpos
[6] (pointer) perform (RPLPTR base+index*2 value)
[7] (float) unbox float value, deposit 32 bits at base+index*2
[11.] (double-pointer) same as hash
[12.] (mixed) same as hash
[not required; not implemented]

#	name	len-1	stk level effect	UFN table entry
53	RPLCHARCODE	0	-2	RPLCHARCODE

[SPECIFICATION INCOMPLETE]

[not required; not implemented]

#	name	len-1	stk level effect	UFN table entry
54	EVAL	0	0	\EVAL

```
takes single argument ARG
If ARG=NIL, T, or smallp, return ARG
If ARG is an atom, attempt free variable lookup:
If bound, return value
        If top value is not NOBIND (atom #1), return top value
else ufn-punt
[optional: if ARG is FIXP, FLOATP, return ARG]
[optional: if ARG is LISTP, punt to \EVALFORM (atom 370q)]
else ufn-punt
[not required; in Dorado, 4K]
```

#	name	len-1	stk level effect	UFN table entry
55	(was EVALV)			

#	name	len-1	stk level effect	UFN table entry
56	TYPECHECK.N	1	0	\TYPECHECK.UFN

identical to DTEST; only UFNs different

#	name	len-1	stk level effect	UFN table entry
57	STKSCAN	0	0	\STKSCAN

TOS is VAR.
If TOS is not litatom, punt.
Returns 24 bit pointer to cell where VAR is bound.

Note: must check VAR=NIL, and return pointer to NIL's value cell. (Free variable lookup algorithm fails if given NIL, at least on Dorado.)

If variable was bound on stack, the value returned will be a pointer into stack space. If variable is not bound, value will be pointer to top level value cell.

[not required; in Dorado (I think), not in DLion? In Maiko emulator]

#	name	len-1	stk level effect	UFN table entry
60	BUSBLT	1	-3	\BUSBLT.UFN

Talks to the BusMaster peripheral adapter.
Alpha bytes:

0	WORDSOUT
1	BYTESOUT
2	BYTESOUTSWAPPED
3	NYBBLESOUT
4	WORDSIN
5	BYTESIN
6	BYTESINSWAPPED
7	NYBBLESINSWAPPED

[not required; in 12K only]

#	name	len-1	stk level effect	UFN table entry
61	MISC8	1	-7	\MISC8.UFN

Miscellaneous opcode for operations needing 8 args.

Alpha bytes:

Alpha	name	function
0	IBLT1 - special-purpose halftone-drawing routine for spectrogram creation	
1	IBLT2 - ditto	

[not required; in 12K only]

#	name	len-1	stk level effect	UFN table entry
62	UBFLOAT3	1	-2	\UNBOXFLOAT3

in 12K only

Alpha bytes:

0	POLY
1	3X3
2	4X4
3	133
4	331
5	144
6	441

for matrix multiply, polynomial evaluation
alpha byte 7: Unboxed ASET

#	name	len-1	stk level effect	UFN table entry
63	TYPEMASK.N	1	0	\TYPEMASK.UFN

similar to TYPEP, except checks if high byte of type table AND with alpha is non-zero, returns TOS if so, NIL otherwise.

#	name	len-1	stk level effect	UFN table entry
64	PROLOGREADPTR			
65	PROLOGREADTAG			
66	PROLOGWRITETAGPTR			
67	PROLOGWRITE0PTR			
70	PSEUDOCOLOR			
72	EQL			

#	name	len-1	stk level effect	UFN table entry
73	DRAWLINE	0	-8	\DRAWLINE.UFN

takes 8 (!) args from top of stack, does line draw inner loop

#	name	len-1	stk level effect	UFN table entry
74	STORE.N	1	0	\STORE.N.UFN

takes quantity at TOS and stores it at TOS-alpha.

#	name	len-1	stk level effect	UFN table entry
75	COPY.N	1	1	\COPY.N.UFN

pushes quantity at (TOS-alpha/2). COPY.N 0 = COPY

#	name	len-1	stk level effect	UFN table entry
76	RAID	0	0	RAID

[used only for UFN]

#	name	len-1	stk level effect	UFN table entry
77	\RETURN			

used only for UFN for LLBREAK

#	name	len-1	stk level effect	UFN table entry
100-106	IVAR	0	1	

push IVAR#(opcode-100) [required]

#	name	len-1	stk level effect	UFN table entry
107	IVARX	1	1	

push IVAR#alpha
[required]

#	name	len-1	stk level effect	UFN table entry
110-116	PVAR	0	1	

push PVAR#(opcode-110)
[required]

#	name	len-1	stk level effect	UFN table entry
117	PVARX	1	1	

push PVAR#(alpha)
[required]

#	name	len-1	stk level effect	UFN table entry
120-126	FVAR	0	1	
127	FVARX	1	1	

Push the indicated FVAR.
[required]

#	name	len-1	stk level effect	UFN table entry
130-136	PVAR_	0	0	
137	PVARX_	1	0	

Set the indicated PVAR from tos, do not pop.
[required]

#	name	len-1	stk level effect	UFN table entry
140	GVAR	2	1	

Push @(VALSPACE+2*(alpha,beta))

[required; may want to change to check if high order bit on, and UFN]

#	name	len-1	stk level effect	UFN table entry
141	ARG0	0	0	\ARG0

check TOS smallp, call UFN if not
check TOS between 1 and #args in current function
replace TOS with value of Ith variable, counting from 1

[to do range check, must fetch flags; if not fast, fetch BLINK.
#args is computable from difference of BLINK and IVAR]

[not required; not implemented yet]

#	name	len-1	stk level effect	UFN table entry
142	IVARX_	1	0	

store TOS as new value of IVAR alpha
[required]

#	name	len-1	stk level effect	UFN table entry
143	FVARX_	1	0	

free variable assignment. When value cell is global, perform GVAR_ operation
[can call \SETFREEVAR.UFN (atom# ???) instead]

#	name	len-1	stk level effect	UFN table entry
144	COPY	0	1	

push TOS again
[required]

#	name	len-1	stk level effect	UFN table entry
145	MYARGCOUNT	0	1	\MYARGCOUNT

Push as a smallpos the number of arguments in current frame.
See ARG0. (probably should use common subroutine)
[not required; not implemented]

#	name	len-1	stk level effect	UFN table entry
146	MYALINK	0	1	

Returns stack-index of beginning of ALINK of current frame.
This pushes the "ALINK" field of the current frame, with the low bit turned off less ALINK.OFFSET (= 12Q).
[required]

#	name	len-1	stk level effect	UFN table entry
147	ACONST	2	1	

Push {0, (alpha,beta)}
[required]

#	name	len-1	stk level effect	UFN table entry
150	'NIL	0	1	
151	'T	0	1	
152	'0	0	1	
153	'1	0	1	

Push the indicated constant.
[required]

#	name	len-1	stk level effect	UFN table entry
154	SIC	1	1	
155	SNIC	1	1	
156	SICX	2	1	

Push:
alpha as a smallposp,
alpha as a smallneg (extend leftward with 1's),
(alpha,beta) as smallposp, respectively.
[required]

#	name	len-1	stk level effect	UFN table entry
157	GCONST	3	1	

Push {alpha, (beta,gamma)}

[required]

#	name	len-1	stk level effect	UFN table entry
160	ATOMNUMBER	2	1	

same as SICX. Different opcode for benefit of code walkers.
[required]

#	name	len-1	stk level effect	UFN table entry
161	READFLAGS	0	0	\READFLAGS

TOS is a virtual page# as a smallposp
TOS _ virtual memory flags of that page, as a smallposp
Flags are:
bit 0: referenced
bit 2: write-protect
bit 3: dirty
Vacant is denoted write-protect + dirty
[This is the same as XNovaOp ReadFlags, with AC0 -> loloc[TOS]]
[required]

#	name	len-1	stk level effect	UFN table entry
162	READRP	0	0	\READRP

TOS is a virtual page# as a smallposp
TOS _ the corresponding real page, as a smallposp
[This is the same as XNovaOp ReadRP, with AC0 -> loloc[TOS]]
[required]

#	name	len-1	stk level effect	UFN table entry
163	WRITEMAP	0	-2	\WRITEMAP

TOS-2 is a virtual page# as a smallposp
TOS-1 is a real page as a smallposp
TOS is a word of flags as a smallposp
Make the indicated virtual page# be associated with the given
real page, with status flags. Real page is immaterial if flags = VACANT
Return the virtual page #
[This is the same as XNovaOp SetFlags, with AC0 -> loloc[TOS-2],
AC1 -> loloc[TOS-1], AC2 -> loloc[TOS]]
[*not yet in Dorado]

#	name	len-1	stk level effect	UFN table entry
164	READPRINTERPORT	0	+1	\READPRINTERPORT

TOS _ current value from printer port, as a smallposp
Ufn if machine cannot do this.
[not in 4k]

#	name	len-1	stk level effect	UFN table entry
165	WRITEPRINTERPORT		0	0 \WRITEPRINTERPORT

Printer _ TOS, interpreted as a smallposp
Ufn if machine cannot do this.
[not in 4k]

#	name	len-1	stk level effect	UFN table entry
166	PILOTBITBLT	0	-1	\PILOTBITBLT

Performs Pilot-style bitblt.
TOS is constant zero, which can be used for maintaining state.
TOS-1 is a pointer to a bitblt table, which is 16-aligned.
[not required, implemented]

#	name	len-1	stk level effect	UFN table entry
167	RCLK	0	0	\RCLKSUBR

Store into words pointed to by TOS the processor clock [up to 32 bits, left justified]. [required]

#	name	len-1	stk level effect	UFN table entry
170	MISC1	1	0	\MISC1.UFN
171	MISC2	1	-1	\MISC2.UFN

These are miscellaneous opcodes that dispatch on alpha to provide infrequent and/or machine-specific operations. To save microcode space (currently), the two opcodes share the same dispatch table, i.e., the alpha's do not overlap. There are two opcodes principally so that there can be a reasonable ufn handler: MISC1 takes 1 arg, MISC2 takes 2. Current values for alpha:

0 STARTIO[bits] Currently only for Dolphin ethernet. Perform the "StartIO" function with bits given as smallp TOS. (Resets Ethernet to known quiet state).

1 INPUT[devreg]Perform input from some device. TOS is smallp device register specification (on Dolphin: 4 bits of task, 4 bits of device reg; on DLion: 4 bits absolute). Returns TOS = smallp value input from device.

DLion codes:
for INPUT {alpha = 1 mod 4}
TOS = 00 mod 16, _ EIData
TOS = 01 mod 16, _ EStatus
TOS = 02 mod 16, _ KIData
TOS = 03 mod 16, _ KStatus
TOS = 04 mod 16, _ uSTATE
TOS = 05 mod 16, _ MStatus
TOS = 06 mod 16, _ KTest
TOS = 07 mod 16, MP code 9122
TOS = 08 mod 16, _ Version
TOS = 09 mod 16, <12K> _ BusExt L <4K> MP code 9122
TOS = 10 mod 16, <12K> _ BusExt M <4K> MP code 9122
TOS = 11 mod 16, <12K> _ uFLmode <4K> MP code 9122
TOS = 12 mod 16, MP code 9122
TOS = 13 mod 16, MP code 9122
TOS = 14 mod 16, MP code 9122
TOS = 15 mod 16, MP code 9122

2 OUTPUT[value, devreg] Perform output to some device. TOS is smallp device register spec as with INPUT; TOS-1 is the smallp value to output.

for DLion:
for OUTPUT {alpha = 2 mod 4}
TOS = 00 mod 16, <12K> BusExt L _ <4K> IOPOData _
TOS = 01 mod 16, IOPCtl _
TOS = 02 mod 16, <12K>uFLmode _ <4K> KOData _
TOS = 03 mod 16, KCtl _
TOS = 04 mod 16, EOData _
TOS = 05 mod 16, EICtl _
TOS = 06 mod 16, DCtl _
TOS = 07 mod 16, uBBTime _ {display rate}
TOS = 08 mod 16, uLispOptions _
TOS = 09 mod 16, PCtl _
TOS = 10 mod 16, MCtl _
TOS = 11 mod 16, <12K> BusExt M _ <4K> MP code 9120
TOS = 12 mod 16, EOCtl _
TOS = 13 mod 16, KCmd _
TOS = 14 mod 16, <12K> PPort _ <4K> MP code 9120
TOS = 15 mod 16, POData _

9 Dorado only, RWMUFMAN

#	name	len-1	stk level effect	UFN table entry

```
172       RECLAIMCELL 0          0                    \GCRECLAIMCELL
```

Check type of TOS; let DTD be pointer to DTD of this type
If not LISTP then punt
Reclaim list:

```
        code_PTR:cdrcode
        if (code and 200q) = 0 then punt        [or optional: if code = 0 then punt]
        FreeListCell(PTR)
        val_ deleteref(PTR:carfield)             * deleteref CAR
        if code # \CDR.NIL
          then PTR_PTR:pagebase + (code lsh 1) * point to cdr or lvcdr
                [if (code and 200q) = 0          * optional
                  then FreeListCell(PTR)                 * cdr indirect--free cell
                        PTR_ GetBasePtr(PTR)]
                if deleteref(PTR)                        * deleteref CDR
                  then val_PTR
        return val
FreeListCell(PTR):
        PAGE _ address of PTR's page
        if PAGE:Nextpage < 0 then punt          * only when page was full
        PTR:cdrcode _ PAGE:nextcell
        PAGE:nextcell _ word# of PTR
        PAGE:count _ PAGE:count + 1
```

How to reclaim other types, roughly (needs type table change):

```
        if Type bit "ok to reclaim" is off, call UFN
        store DTD:FREELST in first two words of DATUM
        store DATUM in DTD:FREELST
```

[not required; implemented for Listp on D0, non-listp on Dorado?, ? for 12K]

#	name	len-1	stk level effect	UFN table entry
173	GCSCAN1	0	0	\GCSCAN1

scan HTMAIN from (TOS)-1 to 0 for a cell with
collision bit on or else stack bit & reference cnt both are 0
if none found, return NIL
else return new index.

note: design allows NWWInterrupts to be processed
note: can actually perform GCRECLAIMCELL on the
cell indicated if stack bit off and ref cnt=0)

[not required; in all]

#	name	len-1	stk level effect	UFN table entry
174	GCSCAN2	0	0	\GCSCAN2

similar to GCSCAN1, but scan for word
with collision bit on or stack bit on.
Note: can optionally turn stack bit off, check if
count is 1 and zero entry, continue scanning
Note: design allows NWWInterrupts to be processed

[not required; in all]

#	name	len-1	stk level effect	UFN table entry
175	SUBRCALL	2		

Call Bcpl subr number alpha with beta arguments.
The following have some microcode on the DLion:

17'b Raid
15'b Logout
06'b BackGround
11'b DspBout
20'b Pup
22'b SETSCREENCOLOR
23'b ShowDisplay

#	name	len-1	stk level effect	UFN table entry
176	CONTEXT	0	0	\CONTEXTSWITCH

switch to context (TOS).

#	name	len-1	stk level effect	UFN table entry
177	(was audio)			

[not required; not currently implemented]

#	name	len-1	stk level effect	UFN table entry
200-217	JUMP	0	JUMP	
220-237	FJUMP	0	CJUMP	
240-257	TJUMP	0	CJUMP	
260	JUMPX	1	JUMP	
261	JUMPXX	2	JUMP	
262	FJUMPX	1	CJUMP	
263	TJUMPX	1	CJUMP	
264	NFJUMPX	1	NCJUMP	
265	NTJUMPX	1	NCJUMP	

Assorted jumps. The offset of the jump is given in the succeeding bytes, sign-extended to the left in the case of the single-byte offsets. The offset is relative to the start of the instruction. The opcodes with implicit offset run from +2 thru +21q.
JUMP* are unconditional.
FJUMP* and TJUMP* perform the jump only if TOS is NIL or non-NIL, respectively.
NFJUMPX and NTJUMPX perform the jump only if TOS is NIL or non-NIL, respectively. Additionally, they pop the stack only if the jump is not taken.
[required]

#	name	len-1	stk level effect	UFN table entry
266	AREF1	0	-1	%AREF1

Perform a one-dimensional array access:

(AREF1 array index)

1.) Check that array is a oned-array -- if not punt

2.) Check that 0 <= index < total size for array

3.) Compute (index + offset for array)

4.) Extract base, and type number -- and pass base, type number, index + offset to array-read subroutine and return result on top of stack.

#	name	len-1	stk level effect	UFN table entry
267	ASET1	0	-2	%ASET1

Perform a one-dimensional array set:

(ASET1 new-value array index)

1.) Check that array is a oned-array -- if not punt

2.) Check that 0 <= index < total size for array

3.) Compute (index + offset for array)

4.) Check array not read-only

5.) Extract base, and type number -- and pass newvalue, base, type number, index + offset to array-write subroutine and return newvalue on top of stack.

#	name	len-1	stk level effect	UFN table entry
270-276	PVAR_^	0	-1	

Store TOS into indicated PVAR, pop stack.
[required]

#	name	len-1	stk level effect	UFN table entry
277	POP	0	-1	

Pop stack.
[required]

#	name	len-1	stk level effect	UFN table entry
300	POP.N	1	(variable)	

POP (alpha+1) elements off top of stack, POP.N 0 = POP, POP.N 1 = POP POP, etc.

#	name	len-1	stk level effect	UFN table entry
301	ATOMCELL.N	1	0	

if TOS is atom (0,,low), then replace with (alpha,,low+low), with carry into alpha. This is used for getting the PList, Def, Val cell of litatoms. If TOS HI is not 0, call UFN. This will allow assigning definitions, plists and values to non-litatoms.

#	name	len-1	stk level effect	UFN table entry
302	GETBASEBYTE	0	-1	\GETBASEBYTE

Retrieve byte at offset TOS from (TOS-1).

#	name	len-1	stk level effect	UFN table entry
303	INSTANCEP	2	0	\INSTANCEP.UFN

return T if typename is subtype of (alpha,beta), else return NIL.
(typename is word 0 of type's DTD; DTD is DTDBase+(type# lsh 4) not locked down
supertype is word 15 of DTD, 0 means no supertype)
[currently only in 12k Dandelion]

#	name	len-1	stk level effect	UFN table entry
304	BLT	0	-2	\BLT

(BLT destinationaddr sourceaddr nwords)
Move nwords from source to destination. If nwords < prespecified constant (currently 10q), then operation is uninterruptable, else must be prepared to service interrupts. On page fault or interrupt, update stack according to how much is moved, and back up pc. Words are moved right to left (high addresses to low), if it makes a difference. Result is unspecified.

#	name	len-1	stk level effect	UFN table entry
305	MISC10	2	-9	\MISC10.UFN

Perform miscellaneous operation on 10 arguments.

alpha	operation
0	PIXELBLT

[not required; in 12k only]

#	name	len-1	stk level effect	UFN table entry
306 (unused)				

#	name	len-1	stk level effect	UFN table entry
307	PUTBASEBYTE	0	-2	\PUTBASEBYTE

Store TOS at offset TOS-1 from (TOS-2), punting if TOS is not smallposp.
Currently ucode punts and ufn errors if offset isn't a smallp.

#	name	len-1	stk level effect	UFN table entry
310	GETBASE.N	1	0	\GETBASE

TOS _ @(TOS+alpha) as a smallposp.

#	name	len-1	stk level effect	UFN table entry
311	GETBASEPTR.N			

TOS _ 24-bit pointer @(TOS+alpha).

#	name	len-1	stk level effect	UFN table entry
312	GETBITS.N.FD			

take 1 arg on stack (PTR) and 2 bytes (n, fd). fetches the "field" fd from the word PTR + n. fd is a mesa field descriptor: the left 4 bits is the number of the "first" bit of the field, while the right 4 bits is the width of the field-1. E.g., 0:17 is the full word, 0:0 is the leftmost bit.

#	name	len-1	stk level effect	UFN table entry
313	Unused			

#	name	len-1	stk level effect	UFN table entry
314	CMLEQUAL	0	-1	CL:EQUAL

Takes two arguments off the stack and performs some cases of the cl:equal predicate. Punts if either argument is a not an immediate datum or a number.
[not required, not implemented on 4K and Dorado]

#	name	len-1	stk level effect	UFN table entry
315	PUTBASE.N	1	-1	\PUTBASE.UFN

Store TOS as word at location (TOS-1)+alpha
Pop (Return TOS-1).
Punt if TOS not smallposp. Note that UFN will specify extra byte for punt.

#	name	len-1	stk level effect	UFN table entry
316	PUTBASEPTR.N	1	-1	\PUTBASEPTR.UFN

Takes (PTR, NEWVAL) on stack, leaves PTR on stack, stores NEWVAL at PTR+N . (note: no punt case)

#	name	len-1	stk level effect	UFN table entry
317	PUTBITS.N.FD	2	-1	\PUTBITS.UFN

Takes (PTR, NEWVAL) on stack, stores bits of NEWVAL at FD field of PTR+N. Returns PTR.
Punt (UFN) if NEWVAL is not smallposp.

#	name	len-1	stk level effect	UFN table entry
320	ADDBASE	0	-1	\ADDBASE
321	VAG2	0	-1	\VAG2
322	HILOC	0	0	
323	LOLOC	0	0	

as before

#	name	len-1	stk level effect	UFN table entry
324	PLUS2	0	-1	PLUS

325	DIFFERENCE	0	-1	DIFFERENCE
326	TIMES2	0	-1	TIMES
327	QUOTIENT	0	-1	QUOTIENT

(same as I- versions, except UFN different. Optionally perform as F- opcode if one of arguments is floating.)

#	name	len-1	stk level effect	UFN table entry
330	IPLUS2	0	-1	\SLOWIPLUS2
331	IDIFFERENCE	0	-1	\SLOWIDIFFERENCE
332	ITIMES2	0	-1	\SLOWITIMES2
333	IQUOTIENT	0	-1	IQUOTIENT
334	IREMAINDER	0	-1	IREMAINDER

unbox TOS & TOS-1
(if SmallPos then 0,,loloc, if SmallNeg then -1,,loloc,
typetest if FIXP then fetch 32 bit quantity)
perform 32x32 operation, and then
if overflow occurs, punt
    [used to say: call OFLOWMAKENUMBER (atom ???)
    with result mod 2^32 as two 16 bit smallposps.
    This can't work; what did we mean?]
If no overflow:
if hi part 0, return SmallPosHi,,lo
if hi part -1, return SmallNegHi,,lo
else need to return large integer. Two choices:
1) set up as if in call to MAKENUMBER (atom ???) with 2 args being
Hi and Lo part of result, as smallposps; or
2) Perform CREATECELL of type FIXP, and then store results
in generated box; return new box
[only smallpos x smallpos required on IPLUS, IDIFFERENCE;
Current implementation status:
    Only smallpos x smallpos on ITIMES in both microcodes
    only smallpos/smallpos for REMAINDER, QUOTIENT in Dorado]

#	name	len-1	stk level effect	UFN table entry
335	IPLUS.N	1	0	\SLOWIPLUS2

add TOS+alpha

#	name	len-1	stk level effect	UFN table entry
336	IDIFFERENCE.N	1	0	\SLOWIDIFFERENCE

subtract TOS-alpha

#	name	len-1	stk level effect	UFN table entry
337	unused			

#	name	len-1	stk level effect	UFN table entry
340	LLSH1	0	0	\SLOWLLSH1
341	LLSH8	0	0	\SLOWLLSH8
342	LRSH1	0	0	\SLOWLRSH1
343	LRSH8	0	0	\SLOWLRSH8

unbox TOS, perform 32 bit operation and box results
as with 2 arg fns
[smallposp -> smallposp required, can UFN in other cases]

#	name	len-1	stk level effect	UFN table entry
344	LOGOR2	0	-1	\SLOWLOGOR2

345	LOGAND2	0	-1	\SLOWLOGAND2
346	LOGXOR2	0	-1	\SLOWLOGXOR2

see IPLUS etc above
[smallposp -> smallposp required, can UFN in other cases]
[32x32 bit implemented in Dorado, D0]

#	name	len-1	stk level effect	UFN table entry
347	LSH	0	-1	LSH

shift TOS-1 arithmetically by TOS.

#	name	len-1	stk level effect	UFN table entry
350	FPLUS2	0	-1	FPLUS2
351	FDIFFERENCE	0	-1	FDIFFERENCE
352	FTIMES2	0	-1	FTIMES2
353	FQUOTIENT	0	-1	FQUOTIENT

[not required; in Dorado, 12K]

#	name	len-1	stk level effect	UFN table entry
354	UBFLOAT2	1	-1	\UNBOXFLOAT2

alpha bytes:
0 ADD x+y
1 SUB x-y
2 ISUB y-x (currently unused)
3 MULT x*y
4 DIV x/y
5 GREAT x>y (returns T/NIL rather than unboxed floating)
6 MAX (max x y) currently unused
7 MIN (min x y) currently unused
8 REM (x remainder y), i.e. x-(floor x/y)*y
9 (UBAREF A I)
Same as AREF1, except that this one returns an unboxed number
implementations: Dorado has GREAT only 12K has all but REM

#	name	len-1	stk level effect	UFN table entry
355	UBFLOAT1	1	0	\UNBOXFLOAT1

alpha byte:
0 BOX (tos -> floating box (tos))
1 UNBOX (tos -> floating unbox (tos), float if FIXP)
2 ABS (currently unused)
3 NEGATE (currently unused)
implemented all on 12K

#	name	len-1	stk level effect	UFN table entry
356	AREF2	0	-2	%AREF2

Perform a two-dimensional array access:

(AREF2 array i j)

1.) Check that array is a twod-array -- if not punt

2.) Check that 0 <= i < bound0

3.) Check that 0 <= j < bound1

4.) Compute (j + i * bound1)

5.) Extract base, and type number -- and pass base, type number, (j + i * bound1) to array-read subroutine and return result on top of stack

#	name	len-1	stk level effect	UFN table entry
357	ASET2	0	-3	%ASET2

Perform a two-dimensional array set:
(ASET2 newvalue array i j)

1.) Check that array is a twod-array -- if not punt

2.) Check that 0 <= i < bound0

3.) Check that 0 <= j < bound1

4.) Compute (j + i * bound1)

5.) Check array not read-only

6.) Extract base, and type number -- and pass base, type number, (j + i * bound1) to array-write subroutine and return newvalue on top of stack.

#	name	len-1	stk level effect	UFN table entry
360	EQ	0	-1	

return T or NIL if (tos)=(tos-1)

#	name	len-1	stk level effect	UFN table entry
361	IGREATERP	0	-1	\SLOWIGREATERP
362	FGREATERP	0	-1	FGREATERP

[IGREATERP required; FGREATERP not implemented]

#	name	len-1	stk level effect	UFN table entry
363	GREATERP	0	-1	GREATERP

Same as IGREATERP (see PLUS, etc)
[not required]

#	name	len-1	stk level effect	UFN table entry
364	EQUAL	0	-1	EQUAL

If args are EQ, return T
If either arg is litatom, return NIL
else call UFN
[not required; not implemented]

#	name	len-1	stk level effect	UFN table entry
365	MAKENUMBER	2	-1	MAKENUMBER

TOS-1 and TOS are smallposp's denoting the hi and lo halves of a 32-bit number.
Return a fixp that represents it:
  If loloc[TOS-1] = 0
    then return SmallPl,,loloc[TOS]
   elseif loloc[TOS-1] = 177777q
    then return SmallNeg,,loloc[TOS]
   else CREATECELL[\FIXP]
Store loloc[TOS-1] and loloc[TOS] as its hi and lo halves
return the new cell
[implemented on 4K, Dorado]

#	name	len-1	stk level effect	UFN table entry
366	BOXIPLUS	0	-1	\BOXIPLUS
367	BOXIDIFFERENCE	0	-1	\BOXIDIFFERENCE

Same as IPLUS2, IDIFFERENCE, except store result @TOS -- first arg is number box (for which optionally check) -- and no overflow check.

#	name	len-1	stk level effect	UFN table entry
370	FLOATBLT	0	-3	\FLOATBLT

Miscellaneous floating point array ops; will eventually be renamed MISC5. Provides access to just about everything the Weitek FP chip does. Operates on two arrays; puts results in a third.
args: (BASE1, BASE2, DEST, N).
Alpha bytes:

0 FLOATWRAP
1 FLOATUNWRAP
2 FLOAT
3 FIX
4 FPLUS
5 FDIFFERENCE
6 FDIFFERENCE
7 (FPLUS (ABS source1) (ABS source2))
10 (ABS (FDIFFERENCE source1 source2))
11 (ABS (FPLUS source1 source2))
20 FTIMES

[not required; implemented on 1108X only]

#	name	len-1	stk level effect	UFN table entry
371	FFTSTEP	0	-1	\FFTSTEP

Takes FFTTABLE as TOS; performs one FFT step thereupon.
[not required; implemented on 1108X only]

#	name	len-1	stk level effect	UFN table entry
372	MISC3	0	-1	\MISC3.UFN

Miscellaneous 3-arg opcode.
Alpha bytes:

0 EXPONENT(source dest n)
source is vector of floatps, dest is vector of words
store exponent of source for n in dest
1 MAGNITUDE
source is a vector of complex, dest is a vector of float
store magnitude of source in dest
2 FLOAT
source is a vector of word, dest is a vector of float
float source & store in dest
3 COMP
source is a vector of float, dest is a vector of complex
spread source into dest, storing 0's.
4 BLKFMAX
5 BLKFMIN
6 BLKFABSMAX
7 BLKFABSMIN
8 FLOATTOBYTE
source is vector of float (must have even number of elements), dest is vector of words
9 ARRAYREAD (base typenumber index)
Dispatch on typenumber and perform a typed get.

[not required; implemented on 1108X only]

#	name	len-1	stk level effect	UFN table entry
373	MISC4	0	-1	\MISC4.UFN

Miscellaneous 4-arg opcode.
Alpha bytes:

0 TIMES
1 PERM
2 PLUS
3 DIFF

4	SEP
6	\BITMAPBIT bitmap x y newvalue (optional)
7	ARRAYWRITE (newvalue base typenumber index) Dispatch on typenumber and perform a typed put

[some confusion on how 0,2,3 different from corresponding TIMES, PLUS, DIF [not required; implemented on 1108X only]

#	name	len-1	stk level effect	UFN table entry
374	reserved on D0, UPCTRACE on Dorado			
375	SWAP	0	0	
376	NOP	0	0	
377	=			

OPCODES IMPLEMENTED IN MICROCODE BY MACHINE

Jan Pedersen
23 June 1986

Key: '09 = 12K Dandetiger
'08 = 4K Dandelion
'00 = Dolphin
'32 = Dorado (as reported by Gwan)
'32L = Dorado (as reported by Larry)
'86-4 = 4K Daybreak
'86-8 = 8K Daybreak
X = Has microcode
P = Prolog microcode set
- = Doesn't have microcode
? = Don't know

Opcodes listed by entry in UFN table

op	'09	'08	'00	'32	'32L	'86-4	'86-8	Name
000	X	X	X	X	-	X	X	-X-
001	X	X	X	X	X	X	X	CAR
002	X	X	X	X	X	X	X	CDR
003	X	X	X	X	X	X	X	LISTP
004	X	X	X	X	X	X	X	NTYPX
005	X	X	X	X	X	X	X	TYPEP
006	X	X	X	X	X	X	X	DTEST {new COERCE}
007	-	-	-	-	-	-	-	CDDR {unused}
010	X	X	X	X	X	X	X	FN0
...	X	X	X	X	X	X	X	...
015	X	X	X	X	X	X	X	FNX
016	X	X	X	X	X	X	X	APPLYFN
017	-	-	-	-	X	-	-	CHECKAPPLY{?}
020	X	X	X	X	X	X	X	RETURN
021	X	X	X	X	X	X	X	BIND
022	X	X	X	X	X	X	X	UNBIND
023	X	X	X	X	X	X	X	DUNBIND
024	X	X	X	X	X	X	X	RPLPTR.N
025	X	X	X	X	X	X	X	GCREF
026	X	-	-	-	-	-	-	ASSOC
027	X	X	X	X	X	X	X	GVAR←
030	X	-	X	X	X	X	X	RPLACA
031	X	-	X	X	X	X	X	RPLACD
032	X	X	X	X	X	X	X	CONS
033	-	-	-	-	-	-	-	{unused}{will be GETPROP}
034	X	-	-	-	-	-	-	FMEMB
035	-	-	-	-	-	-	-	{unused}{will be GETHASH}
036	-	-	-	-	-	-	-	{unused}{will be PUTHASH}
037	X	-	X	X	X	X	X	CREATECELL
040	X	-	X	X	X	X	X	BIN
041	-	-	-	-	-	-	-	BOUT {unused}
042	P	-	-	-	-	-	-	{Prolog}OPFETCHPLUSOPDISP
043	-	-	-	-	-	-	-	{unused}{will be LIST1}
044	-	-	-	-	-	-	-	DOCOLLECT {unused}
045	-	-	-	-	-	-	-	ENDCOLLECT {unused}
046	X	-	X	-	-	X	X	RPLCONS
047	X	-	-	-	-	-	-	LISTGET
050	-	-	-	-	-	-	-	ELT

051	-	-	-	-	-	-	-	NTHCHC
052	-	-	-	-	-	-	-	SETA
053	-	-	-	-	-	-	-	RPLCHARCODE {unused}
054	X	X	X	X	X	X	X	EVAL
055	-	-	-	-	-	-	-	{unused}EVALV
056	X	X	-	-	X	X	X	TYPECHECK
057	X	X	X	?	X	X	X	STKSCAN
060	X	-	-	-	-	-	-	BUSBLT
061	X	-	-	-	-	-	-	MISC8{IBLT1 and IBLT2}
062	X	-	-	-	-	-	-	POLY {Poly; Mat. Multiply}
063	X	X	-	-	X	X	X	TYPEMASK.N
064	P	-	-	-	-	-	-	{Prolog}PROLOGREADPTR
065	P	-	-	-	-	-	-	{Prolog}PROLOGREADTAG
066	P	-	-	-	-	-	-	{Prolog}PROLOGWRITETAGPTR
067	P	-	-	-	-	-	-	{Prolog}PROLOGWRITE0PTR
070	X	-	-	-	-	-	-	PSEUDOCOLOR
071	-	-	-	-	-	X	X	DOVEMISC
072	X	X	-	-	X	-	X	EQL
073	X	-	-	-	-	-	X	DRAWLINE
074	X	X	?	?	X	X	X	STOREN
075	X	X	-	-	X	X	X	COPYN
076	X	X	X	X	-	X	X	RAID
077	-	-	-	-	-	-	-	{unused}RETURN FOR LLBREAK
100	X	X	X	X	X	X	X	IVAR0
...	X	X	X	X	X	X	X	...
107	X	X	X	X	X	X	X	IVARX
110	X	X	X	X	X	X	X	PVAR0
...	X	X	X	X	X	X	X	...
117	X	X	X	X	X	X	X	PVARX
120	X	X	X	X	X	X	X	FVAR0
...	X	X	X	X	X	X	X	...
127	X	X	X	X	X	X	X	FVARX
130	X	X	X	X	X	X	X	PVAR0←
...	X	X	X	X	X	X	X	...
137	X	X	X	X	X	X	X	PVARX←
140	X	X	X	X	X	X	X	GVAR
141	X	X	-	-	X	X	X	ARG0
142	X	X	X	X	X	X	X	IVARX←
143	X	X	X	X	X	X	X	FVARX←
144	X	X	X	X	X	X	X	COPY
145	-	-	-	X	X	-	-	MYARGCOUNT {unused}
146	X	X	X	X	X	X	X	MYALINK
147	X	X	X	X	X	X	X	ACONST
150	X	X	X	X	X	X	X	'NIL
151	X	X	X	X	X	X	X	'T
152	X	X	X	X	X	X	X	'0
153	X	X	X	X	X	X	X	'1
154	X	X	X	X	X	X	X	SIC
155	X	X	X	X	X	X	X	SNIC
156	X	X	X	X	X	X	X	SICX
157	X	X	X	X	X	X	X	GCONST
160	X	X	X	X	X	X	X	ATOMNUMBER
161	X	X	X	X	X	X	X	READFLAGS
162	X	X	X	X	X	X	X	READRP
163	X	X	X	?	-	X	X	WRITEMAP

164	X	-	X	?	X	-	-	READPRINTERPORT
165	X	-	X	?	X	-	-	WRITEPRINTERPORT
166	X	X	X	X	X	X	X	PILOTBITBLT
167	X	X	X	X	X	X	X	RCLK
170	X	X	X	X	X	X	X	MISC1
171	X	X	X	X	-	X	X	MISC2
172	X	X	X	X	-	X	X	RECLAIMCELL
173	X	X	X	X	X	X	X	GCSCAN1
174	X	X	X	X	X	X	X	GCSCAN2
175	X	X	X	X	X	X	X	SUBRCALL
176	X	X	X	X	X	X	X	CONTEXT
177	-	-	?	-	-	-	-	{unused}AUDIO
200	X	X	X	X	X	X	X	JUMP00
...	X	X	X	X	X	X	X	...
207	X	X	X	X	X	X	X	JUMP07
210	X	X	X	X	X	X	X	JUMP10
...	X	X	X	X	X	X	X	....
217	X	X	X	X	X	X	X	JUMP17
220	X	X	X	X	X	X	X	FJUMP00
...	X	X	X	X	X	X	X	...
227	X	X	X	X	X	X	X	FJUMP07
230	X	X	X	X	X	X	X	FJUMP10←
...	X	X	X	X	X	X	X	...
237	X	X	X	X	X	X	X	FJUMP17←
240	X	X	X	X	X	X	X	TJUMP00
...	X	X	X	X	X	X	X	...
247	X	X	X	X	X	X	X	TJUMP07
250	X	X	X	X	X	X	X	TJUMP10
...	X	X	X	X	X	X	X	...
257	X	X	X	X	X	X	X	TJUMP17
260	X	X	X	X	X	X	X	JUMPX
261	X	X	X	X	X	X	X	JUMPXX
262	X	X	X	X	X	X	X	FJUMPX
263	X	X	X	X	X	X	X	TJUMPX
264	X	X	X	X	X	X	X	NFJUMPX
265	X	X	X	X	X	X	X	NTJUMPX
266	X	-	-	-	-	-	-	ARRAYINDEX1
267	X	-	-	-	-	-	-	ARRAYINDEX2
270	X	X	X	X	X	X	X	PVAR0←
...	X	X	X	X	X	X	X	...
276	X	X	X	X	X	X	X	PVAR6←
277	X	X	X	X	X	X	X	POP
300	X	X	-	-	-	X	X	POP.N
301	X	X	-	-	-	X	X	ATOMCELL.N
302	X	X	X	X	X	X	X	GETBASEBYTE
303	-	-	-	-	-	-	-	{unused}
304	X	X	X	X	X	X	X	BLT
305	X	-	-	-	-	-	-	PIXELBLT
306	-	-	-	-	-	-	-	{unused}
307	X	X	X	X	X	X	X	PUTBASEBYTE
310	X	X	X	X	X	X	X	GETBASE.N
311	X	X	X	X	X	X	X	GETBASEPTR.N
312	X	X	X	X	X	X	X	GETBITS.N.FD

313	-	-	-	-	-	-	-	{unused}{new GETBASEFIXP}
314	-	-	-	-	-	-	-	{unused}{new PUTBASEFIXP}
315	X	X	X	X	X	X	X	PUTBASE.N
316	X	X	X	X	X	X	X	PUTBASEPTR.N
317	X	X	X	X	X	X	X	PUTBITS.N.FD
320	X	X	X	X	X	X	X	ADDBASE
321	X	X	X	X	X	X	X	VAG2
322	X	X	X	X	X	X	X	HILOC
323	X	X	X	X	X	X	X	LOLOC
324	X	X	X	X	X	X	X	PLUS2{see notes}
325	X	X	X	X	X	X	X	DIFFERENCE{see notes}
326	X	X	X	X	X	X	X	TIMES2{see notes}
327	X	X	X	X	X	X	X	QUOTIENT{see notes}
330	X	X	X	X	X	X	X	IPLUS2{see notes}
331	X	X	X	X	X	X	X	IDIFFERENCE{see notes}
332	X	X	X	X	X	X	X	ITIMES2{see notes}
333	X	X	X	X	X	X	X	IQUOTIENT{see notes}
334	X	X	X	X	X	X	X	IREMAINDER{see notes}
335	-	-	-	-	-	X	X	{unused}{IPLUS.N}
336	-	-	-	-	-	X	X	{unused}{IDIFFERENCE.N}
337	-	-	-	-	-	-	-	{unused}
340	X	X	X	X	X	X	X	LLSH1{see notes}
341	X	X	X	X	X	X	X	LLSH8{see notes}
342	X	X	X	X	X	X	X	LRSH1{see notes}
343	X	X	X	X	X	X	X	LRSH8{see notes}
344	X	X	X	X	X	X	X	LOGOR2{see notes}
345	X	X	X	X	X	X	X	LOGAND2{see notes}
346	X	X	X	X	X	X	X	LOGXOR2{see notes}
347	-	-	-	-	-	-	-	{unused}{new ALSH}
350	X	-	X	X	X	-	X	FPLUS2
351	X	-	X	X	X	-	X	FDIFFERENCE
352	X	-	X	X	X	-	X	FTIMES2
353	X	-	X	X	X	-	X	FQUOTIENT
354	X	-	-	-	-	-	-	UBFLOAT2 {UFADD, UFSUB, UFISUB, UFMULT, UFDIV, UGREAT, UMAX, UMIN, UREM}
355	X	-	-	-	X	-	-	UBFLOAT1 {UTOB, BTOU, UABS, UNEG, UFIX}
356	X	-	-	-	-	-	-	ARRAYREAD{GENERAL,UNBOXED}
357	X	-	-	-	-	-	-	ARRAYWRITE{GENERAL,UNBOXED}
360	X	X	X	X	X	X	X	EQ
361	X	X	X	X	X	X	X	IGREATERP
362	X	-	X	X	X	-	X	FGREATERP
363	X	X	X	X	X	X	X	GREATERP
364	X	X	?	?	X	-	X	EQUAL
365	X	-	X	X	X	-	X	MAKENUMBER
366	X	-	X	X	X	-	-	BOXIPLUS
367	X	-	X	X	X	-	-	BOXIDIFFERENCE
370	-	-	-	-	-	-	-	MISC5
371	X	-	-	-	-	-	-	FFTSTEP
372	X	-	-	-	-	-	-	MISC3 {Floating Point Array ops: EXP, MAG, FLOAT, COMPLEX, BLKMAX, BLKMIN, BLKABSMAX, BLKABSMIN, FLOATTOBYTE}
373	X	-	-	-	-	-	-	MISC4 {Floating Point Array ops: TIMES, PERM, PLUS,

								DIFFERENCE, MAGIC, BITMAPBIT}
374	-	-	?	-	X	-	-	{reserved for DOLPHIN}
375	X	X	X	X	X	X	X	SWAP
376	X	X	X	X	X	X	X	NOP
377	-	-	-	-	-	-	-	{unused}

notes:

4K microcode:

PLUS2, DIFFERENCE, TIMES2, QUOTIENT will ufn if args not INTEGERS

IPLUS2, IDIFFERENCE will accept FIXP's as arguments, but will ufn if result is not a smallp or smallneg

ITIMES2, IQUOTIENT, IREMAINDER will ufn if both args are not smallp

12K microcode:

PLUS2, DIFFERENCE, TIMES2, QUOTIENT will try floating point if args not INTEGERS

IPLUS2, IDIFFERENCE will accept FIXP's as arguments, and box the result if it is not a smallp or smallneg

ITIMES2, IQUOTIENT, IREMAINDER will ufn if both args are not smallp

Date: 24 Jun 86 14:36 PDT
From: Masinter.pa
Subject: Opcode survey
To: Pedersen.pa
cc: xclispcore^

edits are in bold. We should meet to review what's in microcode and what the priority list should be. I suggest we do this at the end of the Common Lisp status meeting... does anyone have any objection to discussing it then?

OPCODES IMPLEMENTED IN MICROCODE BY MACHINE

Jan Pedersen

Key: '09 = 12K Dandetiger
'08 = 4K Dandelion
'00 = Dolphin
'32 = Dorado (as reported by Gwan)
'32L = Dorado (as reported by Larry)
'86-4 = 4K Daybreak
'86-8 = 8K Daybreak
X = Has microcode
P = Prolog microcode set
- = Doesn't have microcode
? = Don't know

Opcodes listed by entry in UFN table

op	'09	'08	'00	'32	'32L	'86-4	'86-8	Name
000	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-X-** ***nobody implements: causes an error!***
001	X	X	X	X	X	X	X	CAR
002	X	X	X	X	X	X	X	CDR
003	X	X	X	X	X	X	X	LISTP
004	X	X	X	X	X	X	X	NTYPX
005	X	X	X	X	X	X	X	TYPEP
006	X	X	X	X	X	X	X	DTEST {new COERCE}
007	-	-	-	-	-	-	-	CDDR {unused}
010	X	X	X	X	X	X	X	FN0
...	X	X	X	X	X	X	X	...
015	X	X	X	X	X	X	X	FNX
016	X	X	X	X	X	X	X	APPLYFN
017	-	-	-	-	X	-	-	CHECKAPPLY{?} [**not 86-8?**]
020	X	X	X	X	X	X	X	RETURN
021	X	X	X	X	X	X	X	BIND
022	X	X	X	X	X	X	X	UNBIND
023	X	X	X	X	X	X	X	DUNBIND
024	X	X	X	X	X	X	X	RPLPTR.N
025	X	X	X	X	X	X	X	GCREF
026	X	-	-	-	-	-	-	ASSOC [**08 but not 86-8?]**
027	X	X	X	X	X	X	X	GVAR←
030	X	-	X	X	X	X	X	RPLACA
031	X	-	X	X	X	X	X	RPLACD [**which cases?]**
032	X	X	X	X	X	X	X	CONS
033	-	-	-	-	-	-	-	{**used for GETPROP**}
034	X	-	-	-	-	-	-	FMEMB
035	-	-	-	-	-	-	-	{**used for GETHASH**}
036	-	-	-	-	-	-	-	{unused, named PUTHASH}
037	X	-	X	X	X	X	X	CREATECELL

040	X	-	X	X	X	X	X	BIN
041	-	-	-	-	-	-	-	BOUT {unused}
042	P	-	-	-	-	-	-	{Prolog}OPFETCHPLUSOPDISP
043	-	-	-	-	-	-	-	{unused, named LIST1}
044	-	-	-	-	-	-	-	DOCOLLECT {unused}
045	-	-	-	-	-	-	-	ENDCOLLECT {unused}
046	X	-	X	-	-	X	X	RPLCONS
047	X	-	-	-	-	-	-	LISTGET
050	-	-	-	-	-	-	-	ELT
051	-	-	-	-	-	-	-	NTHCHC
052	-	-	-	-	-	-	-	SETA
053	-	-	-	-	-	-	-	RPLCHARCODE {unused}
054	X	X	X	X	X	X	X	EVAL
055	-	-	-	-	-	-	-	{**used for EVALV**}
056	X	X	-	-	X	X	X	TYPECHECK **{unused}**
057	X	X	X	?	X	X	X	STKSCAN
060	X	-	-	-	-	-	-	BUSBLT
061	X	-	-	-	-	-	-	MISC8{IBLT1 and IBLT2}
062	X	-	-	-	-	-	-	POLY {Poly; Mat. Multiply}
063	X	X	-	-	X	X	X	TYPEMASK.N
064	P	-	-	-	-	-	-	{Prolog}PROLOGREADPTR
065	P	-	-	-	-	-	-	{Prolog}PROLOGREADTAG
066	P	-	-	-	-	-	-	{Prolog}PROLOGWRITETAGPTR
067	P	-	-	-	-	-	-	{Prolog}PROLOGWRITE0PTR
070	X	-	-	-	-	-	-	PSEUDOCOLOR
071	-	-	-	-	-	X	X	DOVEMISC
072	X	X	-	-	X	-	X	EQL
073	X	-	-	-	-	-	X	DRAWLINE
074	X	X	?	?	X	X	X	STOREN
075	X	X	-	-	X	X	X	COPYN
076	X	X	X	X	-	X	X	RAID
077	-	-	-	-	-	-	-	{unused}RETURN FOR LLBREAK
100	X	X	X	X	X	X	X	IVAR0
...	X	X	X	X	X	X	X	...
107	X	X	X	X	X	X	X	IVARX
110	X	X	X	X	X	X	X	PVAR0
...	X	X	X	X	X	X	X	...
117	X	X	X	X	X	X	X	PVARX
120	X	X	X	X	X	X	X	FVAR0
...	X	X	X	X	X	X	X	...
127	X	X	X	X	X	X	X	FVARX
130	X	X	X	X	X	X	X	PVAR0←
...	X	X	X	X	X	X	X	...
137	X	X	X	X	X	X	X	PVARX←
140	X	X	X	X	X	X	X	GVAR
141	X	X	-	-	X	X	X	ARG0
142	X	X	X	X	X	X	X	IVARX←
143	X	X	X	X	X	X	X	FVARX←
144	X	X	X	X	X	X	X	COPY
145	-	-	-	X	X	-	-	MYARGCOUNT **{used}**
146	X	X	X	X	X	X	X	MYALINK
147	X	X	X	X	X	X	X	ACONST
150	X	X	X	X	X	X	X	'NIL
151	X	X	X	X	X	X	X	'T
152	X	X	X	X	X	X	X	'0

153	X	X	X	X	X	X	X	'1
154	X	X	X	X	X	X	X	SIC
155	X	X	X	X	X	X	X	SNIC
156	X	X	X	X	X	X	X	SICX
157	X	X	X	X	X	X	X	GCONST
160	X	X	X	X	X	X	X	ATOMNUMBER **(used?)**
161	X	X	X	X	X	X	X	READFLAGS
162	X	X	X	X	X	X	X	READRP
163	X	X	X	?	-	X	X	WRITEMAP
164	X	-	X	?	X	-	-	READPRINTERPORT
165	X	-	X	?	X	-	-	WRITEPRINTERPORT
166	X	X	X	X	X	X	X	PILOTBITBLT
167	X	X	X	X	X	X	X	RCLK
170	X	X	X	X	X	X	X	MISC1
171	X	X	X	X	-	X	X	MISC2
172	X	X	X	X	-	X	X	RECLAIMCELL
173	X	X	X	X	X	X	X	GCSCAN1
174	X	X	X	X	X	X	X	GCSCAN2
175	X	X	X	X	X	X	X	SUBRCALL {used on 08 86?}
176	X	X	X	X	X	X	X	CONTEXT
177	-	-	?	-	-	-	-	{unused}AUDIO
200	X	X	X	X	X	X	X	JUMP00
...	X	X	X	X	X	X	X	...
207	X	X	X	X	X	X	X	JUMP07
210	X	X	X	X	X	X	X	JUMP10
...	X	X	X	X	X	X	X	....
217	X	X	X	X	X	X	X	JUMP17
220	X	X	X	X	X	X	X	FJUMP00
...	X	X	X	X	X	X	X	...
227	X	X	X	X	X	X	X	FJUMP07
230	X	X	X	X	X	X	X	FJUMP10←
...	X	X	X	X	X	X	X	...
237	X	X	X	X	X	X	X	FJUMP17←
240	X	X	X	X	X	X	X	TJUMP00
...	X	X	X	X	X	X	X	...
247	X	X	X	X	X	X	X	TJUMP07
250	X	X	X	X	X	X	X	TJUMP10
...	X	X	X	X	X	X	X	...
257	X	X	X	X	X	X	X	TJUMP17
260	X	X	X	X	X	X	X	JUMPX
261	X	X	X	X	X	X	X	JUMPXX
262	X	X	X	X	X	X	X	FJUMPX
263	X	X	X	X	X	X	X	TJUMPX
264	X	X	X	X	X	X	X	NFJUMPX
265	X	X	X	X	X	X	X	NTJUMPX
266	X	-	-	-	-	-	-	ARRAYINDEX1
267	X	-	-	-	-	-	-	ARRAYINDEX2
270	X	X	X	X	X	X	X	PVAR0←
...	X	X	X	X	X	X	X	...
276	X	X	X	X	X	X	X	PVAR6←
277	X	X	X	X	X	X	X	POP
300	X	X	-	-	-	X	X	POP.N
301	X	X	-	-	-	X	X	ATOMCELL.N

302	X	X	X	X	X	X	X	GETBASEBYTE
303	-	-	-	-	-	-	-	{unused}
304	X	X	X	X	X	X	X	BLT
305	X	-	-	-	-	-	-	PIXELBLT
306	-	-	-	-	-	-	-	{unused}
307	X	X	X	X	X	X	X	PUTBASEBYTE
310	X	X	X	X	X	X	X	GETBASE.N
311	X	X	X	X	X	X	X	GETBASEPTR.N
312	X	X	X	X	X	X	X	GETBITS.N.FD
313	-	-	-	-	-	-	-	{**used for GETBASEFIXP**}
314	-	-	-	-	-	-	-	{**used for PUTBASEFIXP**}
315	X	X	X	X	X	X	X	PUTBASE.N
316	X	X	X	X	X	X	X	PUTBASEPTR.N
317	X	X	X	X	X	X	X	PUTBITS.N.FD
320	X	X	X	X	X	X	X	ADDBASE
321	X	X	X	X	X	X	X	VAG2
322	X	X	X	X	X	X	X	HILOC
323	X	X	X	X	X	X	X	LOLOC
324	X	X	X	X	X	X	X	PLUS2{see notes}
325	X	X	X	X	X	X	X	DIFFERENCE{see notes}
326	X	X	X	X	X	X	X	TIMES2{see notes}
327	X	X	X	X	X	X	X	QUOTIENT{see notes}
330	X	X	X	X	X	X	X	IPLUS2{see notes}
331	X	X	X	X	X	X	X	IDIFFERENCE{see notes}
332	X	X	X	X	X	X	X	ITIMES2{see notes}
333	X	X	X	X	X	X	X	IQUOTIENT{see notes}
334	X	X	X	X	X	X	X	IREMAINDER{see notes}
335	-	-	-	-	-	X	X	{unused}{IPLUS.N}
336	-	-	-	-	-	X	X	{unused}{IDIFFERENCE.N}
337	-	-	-	-	-	-	-	{unused}
340	X	X	X	X	X	X	X	LLSH1{see notes}
341	X	X	X	X	X	X	X	LLSH8{see notes}
342	X	X	X	X	X	X	X	LRSH1{see notes}
343	X	X	X	X	X	X	X	LRSH8{see notes}
344	X	X	X	X	X	X	X	LOGOR2{see notes}
345	X	X	X	X	X	X	X	LOGAND2{see notes}
346	X	X	X	X	X	X	X	LOGXOR2{see notes}
347	-	-	-	-	-	-	-	{unused}{new ALSH}
350	X	-	X	X	X	-	X	FPLUS2
351	X	-	X	X	X	-	X	FDIFFERENCE
352	X	-	X	X	X	-	X	FTIMES2
353	X	-	X	X	X	-	X	FQUOTIENT
354	X	-	-	-	-	-	-	UBFLOAT2 {UFADD, UFSUB, UFISUB, UFMULT, UFDIV, UGREAT, UMAX, UMIN, UREM}
355	X	-	-	-	X	-	-	UBFLOAT1 {UTOB, BTOU, UABS, UNEG, UFIX}
356	X	-	-	-	-	-	-	ARRAYREAD{GENERAL,UNBOXED}
357	X	-	-	-	-	-	-	ARRAYWRITE{GENERAL,UNBOXED}
360	X	X	X	X	X	X	X	EQ
361	X	X	X	X	X	X	X	IGREATERP
362	X	-	X	X	X	-	X	FGREATERP
363	X	X	X	X	X	X	X	GREATERP
364	X	X	?	?	X	-	X	EQUAL
365	X	-	X	X	X	-	X	MAKENUMBER
366	X	-	X	X	X	-	-	BOXIPLUS
367	X	-	X	X	X	-	-	BOXIDIFFERENCE

370	-	-	-	-	-	-	-	MISC5
371	X	-	-	-	-	-	-	FFTSTEP
372	X	-	-	-	-	-	-	MISC3 {Floating Point Array ops: EXP, MAG, FLOAT, COMPLEX, BLKMAX, BLKMIN, BLKABSMAX, BLKABSMIN, FLOATTOBYTE}
373	X	-	-	-	-	-	-	MISC4 {Floating Point Array ops: TIMES, PERM, PLUS, DIFFERENCE, MAGIC, BITMAPBIT}
374	-	-	?	-	X	-	-	{reserved for DOLPHIN}
375	X	X	X	X	X	X	X	SWAP
376	X	X	X	X	X	X	X	NOP
377	-	-	-	-	-	-	-	{unused}

notes:

4K microcode:

PLUS2, DIFFERENCE, TIMES2, QUOTIENT will ufn if args not INTEGERS

IPLUS2, IDIFFERENCE will accept FIXP's as arguments, but will ufn if result is not a smallp or smallneg

ITIMES2, IQUOTIENT, IREMAINDER will ufn if both args are not smallp

12K microcode:

PLUS2, DIFFERENCE, TIMES2, QUOTIENT will try floating point if args not INTEGERS

IPLUS2, IDIFFERENCE will accept FIXP's as arguments, and box the result if it is not a smallp or smallneg

ITIMES2, IQUOTIENT, IREMAINDER will ufn if both args are not smallp

—End of message—

SEdit Linearizer Internal Documentation

Fields in the context:

CurrentNode
The node whose linear form is currently being computed.

LinearPointer
Points to the "next" item in the linear form. This is the item with which the next generated item will be compared, and the item before which it will be inserted if it doesn't match.

LinearPrev
If LinearPrev is a cons, (CDR LinearPrev) is LinearPointer (LinearPrev is one behind LinearPointer in the linear form). Otherwise, it's a node, and (fetch LinearForm of LinearPrev) is LinearPointer. Used to fixup linear form.

CurrentLine
The LineStart most recently generated in the linear form.

CurrentX
The X coordinate at which the next linear item will be displayed.

RightMargin
The right margin for generating the linear form.

CurrentBlock
The LineBlock describing the most recently generated linear items on this line. Reset to FirstBlock at the end of each line displayed.

FirstBlock
The beginning of the LineBlock list. The LineBlocks from FirstBlock to CurrentBlock describe the segment of the linear form between CurrentLine and LinearPointer, indicating which parts are already available in the window for BITBLTing and which will have to be repainted.

Matching?
Means something like: the linear form we're generating has been matching the linear form that was already there (at least since the beginning of CurrentNode)

Below?
T if the linear form we're generating is definitely off the bottom of the screen. NIL if it might have to be displayed. 'new if we're redisplaying from scratch (nothing to BLT).

Visible?
T if we're matching and the bits we're matching are actually on the screen.

RelinearizationTimeStamp
NIL if we're prettyprinting, otherwise incremented by 1 each time we relinearize from the top. Used to determine the validity of cached info in LineStarts

RepaintStart
RepaintLine
RepaintX
If there are no bits to be reused at the end of a line, we postpone displaying it until we find something that needs to be moved (or we get to the end of the window). This can go on for many lines. during this time, RepaintStart records where the painting needs to start from, RepaintLine has the y information, and RepaintX records where the painting will start from. It may always be the case that

```
   RepaintLine = (CAR RepaintStart)
```

```
   RepaintX = (fetch Indent of RepaintLine)
```

Fields in EditNodes:
StartX
The X coordinate at the time this node was linearized. not sure what 0 means (something magic). \\reuse.linear.form seems to think this means that the node is atomic and hasn't been displayed before.

The coordinate system:
The top left corner is (1,-1) (or maybe (0,0)?). Therefore everything's displayed with positive x and negative y (bottom right quadrant).

List Formats

SEdit allows one to specify how a class of forms are to be pretty printed. This is done by defining a list format on a symbol. This causes all forms whose car is this symbol to be displayed with the specified format.

List formats for most Common LISP and Interlisp special forms are provided with SEdit. The source code for these definitions can be found on the Lisp Library Floppy #XXX.

`(def-list-format` *name* {*doc-string*} {*format-name* | `&key` *:args :sublists :inline :miser :last :indent*}`)` [Definer]

Tells SEdit how to prettyprint forms whose CAR is NAME.

Short form

If *FORMAT-NAME* is provided then NAME is defined to be formatted just like *FORMAT-NAME*. If, for example one had defined a list format for `dotimes` one could then define `dolist` to be formatted the same way with:

```
(def-list-format dotimes dolist)
```

Long Form

The keyword arguments have the following meanings:

`:ARGS` -- value should be a list of names of list formats. These formats are assigned to the elements of the list in order starting with the first element (which will be `NAME`). Note that these formats override any formats that would normally be assigned to the elements of the list (based on their first elements). NIL is allowed in the `:ARGS` list, and means do *not* override the format of this element; that is, allow it to be formatted normally. Also, a symbol S is allowed in the :ARGS list if S has earlier been assigned a format; this means to assign S's format to this element. There are also two special keywords allowed as entries in the :ARGS list: :KEYWORD and :RECURSIVE. :KEYWORD means that if the element assigned this format is a symbol then treat it like a keyword, i.e., put it in bold face. (This list uses the convention that all symbols which allow declarations in their body [such as DO and LET] are formatted as keywords.) :RECURSIVE means to assign this element the same format as is being defined; that is, the entire top level format is assigned recursively to this element. This is very useful for formats like :DATA format (see below). If L has more elements than there are entries in the :ARGS list, the last entry in the :ARGS list is repeated for all the extra elements of L. Hint: most :ARGS entries have NIL as their last element. If no :ARGS list is specified, the elements of L get their natural formatting.

:SUBLISTS -- value should be a list of element positions (counting from 1) or T. T means all of the arguments should be parsed as lists even if they are NIL (so NIL will display as () rather than NIL). A list of element position means those element positions will be parsed as lists. For example LET has :SUBLISTS (2) meaning the second element of a form whose first element is LET is a list (i.e., the binding list). DO has :SUBLISTS (2 3), DEFUN has :SUBLISTS (3) and COND has :SUBLISTS T. Default is :SUBLISTS NIL meaning print all NIL args as NIL not ().

:INLINE -- value can be T or NIL (default NIL). If T, the form will go all on one line if it fits. If NIL, the form will be broken across lines at arg boundaries even if it would all fit on one line. For example, OR has :INLINE T and LET has :INLINE NIL.

:MISER -- value can be :ALWAYS, :NEVER, or :TOFIT (default :TOFIT). Specifies when to use miser indentation. The default means use miser indentation if non-miser indentation would force the arguments into miser indentation. **[need to explain what miser mode is]**

:LAST -- value should be a format specification like those in the :ARGS list. This format specification will be applied to the last element of L but *only* if doing so would supercede the last entry in the :ARGS list. In other words, if the last element of L would receive the repeated format from the :ARGS list, it gets the :LAST format instead. This option is really only useful for pathologically formatted forms like Interlisp's SELECTQ.

:indent -- An indentation specification is either a symbol (normally a keyword) or a list. If it's a symbol, it's looked up on the SEDIT::*INDENT-ALIST* (which see) and the SEdit-internal indent specification found there is used. If it’s a list, it consists of some optional keywords (described below) followed by argument group specifications. Each argument group specification is either a number or a list containing a single number. In both formats, the number indicates that that many arguments should be grouped together at a single indentation level. The simple number format means that each of those arguments should go on its own line (they will line up vertically with each other), while the number-in-a-list format means that the arguments in the group can go together on a single line if they fit. The indentation level for each argument group is determined by how many groups follow it in the indentation list. Each group is indented 1 level further in than the group which follows it; thus, the first argument group is indented most, the next one next most, and so on until the last one, which is always indented one step in from lambda-body level.

This is best explained with examples. A simple example is LET, whose indentation specification is (1). This means that LET will be followed by a single distinguished argument group consisting of one element (the binding list) which will be indented one step in from the let body. Another simple example is DO, whose indentation specification is (2). This means that DO will be followed by a single distinguished argument group consisting of two elements (the binding list and the termination clause) which will be indented one step in from the do body. It also means that the bindings and the termination will be required to go on separate lines. Contrast DO with DEFUN, whose indentation specification is ((2)). Like DO’s spec, DEFUN’s spec says there is one group with two members (the name and the lambda-list), but unlike DO’s spec, DEFUN’s spec says that the first two args can go on the same line if they fit there. Finally, consider a possible spec for MULTIPLE-VALUE-BIND of (1 1) which says that the first group consists of one arg (the variable list) and the second group consists of one arg (the form to eval). The form to eval will be indented one step in from the body, and the list of variables will be indented one step in from there.

Note that a group specification of 0 (zero) is allowed: this occupies an indentation step but does not put any arguments at that level. But we do not allow (0) as a group specification since this would not be any different than plain 0 and probably means that the specification is confused in some way.

The keywords allowed at the beginning of an indent specification are:

:BREAK or **:NOBREAK** or **:FIT** -- These specify placement of the first argument in the first group. Default is :FIT, which means put this arg on the same line as the CAR of the form if it fits there in preferred mode, otherwise put it on the next line. Note that if the first arg goes on the same line as the CAR, its placement specifies the indentation level for the entire first group. That way long CARs will move the first group over to the right. (This makes the binding and termination of both DO and DO* line up, for example.) Specifying :NOBREAK means the first arg is forced to go on the same line as the CAR. Specifying :BREAK means the first arg is forced to go on the next line (and thus at the indentation level derived from the number of groups). UNWIND-PROTECT is a good example of using :BREAK to force the first arg onto its own line. Note that you can only specify one of :NOBREAK, :BREAK, or :FIT.

:TAGBODY -- Normally all forms in the body (whether atomic or not) go at the same indent level. Specifying :TAGBODY indicates that atomic body elements (*not* atomic elements of the argument groups) should be exdented to line up with the CAR of the entire list (such as PROG or TAGBODY, which see for examples).

:STEP -- This can be specified as many times as desired and each time increases the indentation of the body (and thus all the argument groups) by one step. If you just want to move some of the groups in but not all of them (and not the body) then use 0 groups at the appropriate place instead of using :STEP at the beginning. :STEP is very useful with :TAGBODY.

By the way, the normal body indentation is taken from the INDENT-BASE field of the LISP edit environment, which is initialized to the width of a capital ‘M’ in the SEdit default font. The normal indentation step is taken from the INDENT-STEP field of the LISP edit environment, which is initialized to twice the width of a capital ‘M’ (that is, twice INDENT-BASE). These defaults are chosen so that, in a fixed-width font, the body of a form lines up two characters in from the ‘(’ of the form, and each argument group line up two characters in from the next one (or the body). If you want non-standard values for either of these parameters, you can change the values in the LISP edit environment and then reinitialize your SEdit formats. Also, if you change font profiles, reinitializing SEdit will fix up the indents appropriately.

SEdit Internal Documentation

The formatting methods for lists

The assign-format, compute-format-values, and linearize methods of lists are now driven by tables encapsulated in list-format objects, allowing easy special formatting for particular lisp forms. This note primarily documents the format of list-format objects, and along the way mentions how they are used to implement those three methods. (note that dot-lists currently aren't handled by this mechanism)

Finding the right list-format

assign-format-list finds an appropriate list-format to control the list's formatting, using one of four options:

> if the list is assigned a format which is a list-format object, use that;
> else if the car of the list can be found in a list-formats table,
>   use the associated object from that table;
> else if the car of the list has a known clispword property,
>   use the clisp list-format
> else use the default list-format

since the same list-format object will be needed for computing width estimates and and linearization, assign-format caches it in the unassigned file. there are two types of list-formats; standard and non-standard. standard list-formats contain information to control the standard list formatting methods. nonstandard list-formats are an escape mechanism for situations where the required formatting is too hairy for the standard methods; they simply provide replacements for the assign-format, compute-format-value, and linearize methods. this is implemented by all list-formats having a non-standard? field, and list-formats whose non-standard? field is t having 3 additional fields: set-format-list, cfv-list, and linearize-list. there's not much else to say about non-standard list-formats, except that at present the only one is the format for clisp. for the rest of this document we'll talk about standard list-formats.

A general rule

several list-format fields contain lists of entries which correspond to the list node's subnodes. these lists all have roughly the same form:

> (*last first second ... nth*)

where *first* is the information to be used for the formatting the first subnode, *second* the information for the second, ..., *nth* the information for the nth and all subsequent nodes, except that *last* is the information to be used for the last (some forms (e.g. `il:selectq`) have special formatting for the very last item). n depends on how many of the nodes need special formatting. `lambda`, for instance, uses lists of the form (a b c a) — thus the first element is formatted using b, the second with c, and all subsequent ones with a. the default list format uses lists of the format (a), since all subnodes are formatted similarly. `il:selectq` uses lists of the form (a b c d) — so b applies to the `il:selectq` atom, c to the evaluated expression, d to each clause, and a to the final otherwise clause.

the other important fact about these lists is that they're blind to comments; comment subnodes are ignored when figuring out which information goes to which subnode, and the comments themselves are formatted by hardwired rules.

Assigning formats

assign-format-list uses the list-formats field of the list-object, which is a list in the format described above. each element of the list is the format to assign to the corresponding subnode (nil, :keyword, or a list-format object — or :recursive, which means to assign this node's list-format to its subnode).

Linearizing

each list-format contains descriptions of two possible presentations of that list — a “preferred” format and a “miser” format, in the list-pformat and list-mformat fields of the list-format. linearization decides which presentation is appropriate, based on the width estimates of this node (see below for a description of where they come from) and the horizontal space available.

the list linearization will always have the form "(" *subnode* <space> *subnode* <space> ... *subnode* ")", where <space> is either a one-space-wide horizontal movement, or a line break (with some indentation) (we assume the list contains no comments for now). thus, the problem of formatting the list reduces to specifying, for each element after the first, whether to space or break, and if you break, how much to indent. using a list in the above format, a specification is given for each subnode after the first as to how this decision is to be made.

these spacing specifications are expressions in a simple language. each expression must at a minimum specify the indentation if a line break is made here. the simplest expressions are just that — an integer, giving the indentation (in pixels) from the opening "(". this expression won’t break unless it has to, but if it does it will use that indentation. to force a break, an expression of the form `(break .` *exp*`)` is used (where *exp* is a nested expression). to line up the presentation of subnodes, spacing specifications can set a tab stop and then later position relative to it. `(set-indent .` *exp*`)` works just like *exp*, except that after it has been determined where this subnode is to be positioned, the tab stop is set at that point. a later spacing specification of `(from-indent .` *exp*`)` means that if we break at this point, the indentation is to be taken relative to the tab stop. note that the order of parts in an expression isn’t important; `(break from-indent . 3)` has the same effect as `(from-indent break . 3)`.

the remaining types of expressions allow the formatting to depend on a variety of conditions, such as whether the previous node’s presentation fit on one line, or whether this node is atomic. they all have the form `(`*condition exp1* `.` *exp2*`)`, where *condition* is one of the atoms below, *exp1* is the spacing specification to be used if the condition holds, and *exp2* is the spacing specification to be used if it doesn’t. (remember that a spacing specification is interpreted to determine the space preceding a subnode; in the list below, "previous node" is the node before the one whose placement is being determined, and "next node" is the node whose placement is being determined).

`prev-inline`:	did the previous subnode’s presentation fit on one line?
`next-inline`:	does the next subnode’s width estimate indicate that it will fit on this line?
`next-preferred`:	does the next subnode’s width estimate indicate that it will fit in preferred format?
`prev-atom`, `next-atom`:	is the previous / next node atomic (i.e. has no subnodes)?
`prev-keyword`, `next-keyword`:	is the previous / next node an atom in the keyword package?
`prev-lambdaword`, `next-lambdaword`:	is the previous / next node a lambda keyword (&aux, &rest, etc)?
`whole-inline`:	do the width estimates of the node being formatted indicate that it will fit on one line?

Formatting comments

the preceding discussion assumed that the list being formatted contained no comments. the formatting of comments is completely automatic (i.e. out of the control of the list-format), primarily to simplify list-formats. they are formatted like other subnodes, except that they don’t use the spacing specifications. single-semi comments are positioned at a fixed horizontal position (at the end of the current line if it isn’t too long, otherwise on a new line). triple-semi comments always start on a new line, at the left margin. double-semi comments are a little trickier; they start at the current tab stop, after the spacing specification for the next node has been interpreted. this requires looking ahead in these cases, and determining what the result of interpreting the space specification will be. it would probably be worth figuring out simpler ways to do this.

subnodes following comments always start on a new line (i.e. their spacing specifications are interpreted as if they began `(break . ...)`).

Computing width estimates

cfv-list must compute two values: the width of this node if its presentation can fit on one line (or nil if it can't), and the width of this node in its "preferred" presentation. the second of these is computed by simulating the linearization process using the preferred spacing specifications, and always assuming the worst when evaluating the prev-inline, next-inline, next-preferred, and whole-inline conditions. the inline width is determined by an even simpler scheme: if all of the subnodes can go inline, and the list-inline? field of the list-format isn't nil, the list can go inline with width equal to the sum of its subnode's inline widths, plus the appropriate blanks and parens.

Node Types and Node Type Methods in SEdit

All SEdit editing operations and presentations are controlled by methods associated with classes of nodes in the edit tree. By defining a new class of nodes with appropriate methods, or modifying an existing class, SEdit may be configured for a wide range of editing tasks. This document describes the programming involved in defining such a class, using the basic Interlisp-D type definitions as examples.

Node Types and the Edit Tree

The edit tree is a data structure maintained by SEdit as a representation of the structure being edited. The tree is initially constructed by parsing the data structure to be edited as a hierarchical data structure. Each node in the tree corresponds to a part of the edited structure, either an instance of a lisp datatype or a combination of several data structures. For instance, atoms and strings are usually represented by separate nodes in the tree, but a sequence of several cons cells may be represented by a single list node. Instances of a single data type may be represented by a variety of node types, depending on the context in which they appear. All editing operations are defined in terms of these nodes. The definitions for editing Interlisp code define ten node types:

1. atoms (actually litatoms and numbers)
2. strings
3. lists (NIL-terminated sequences of cons cells)
4. dotted lists (sequences of cons cells terminated by something other than NIL)
5. CLisp expressions (if, fetch, iteration, etc.)
6. forms (lisp function calls)
7. LAMBDA expressions (also includes LETs, PROGs, etc.)
8. quoted structures (i.e. two element lists whose first element is the atom QUOTE)
9. unknown (any data type other than litatom, number, string, or cons)
10. root (a special node type for the root of the edit tree)

Node types 5 through 8 are special cases of type 3, which are recognized in some contexts.

Creating the Edit Tree

To create the edit tree, SEdit conducts a preorder traversal of the given data structure. At each step, it dispatches on the datatype of the structure and calls the corresponding function, which is responsible for building the rest of the tree by making a call to \\create.node and recursive calls to the parser (\\parse). Each node built will record (among other things)

- the node type (chosen by the parse function)
- the node's super node
- the node's subnodes (one for each recursive call to \\parse)
- the node's depth in the tree
- the structure which was parsed to create this node

Because the correct parsing of a node often depends on contextual information, the parser allows each parse method to pass an argument to its subnodes. For no particularly good reason this argument is called the *parse mode*. The interpretation of the parse mode is up to the individual parse methods. The default mode is NIL; most parse methods ignore any unrecognized mode. In addition to the default mode, the Interlisp code definitions use a couple of other modes (the atoms `Binding`, `BindingList`, and `KeyWord`).

Edit Node Types

The type information SEdit associates with each node is actually a set of methods, i.e. functions to perform various actions on nodes of that class. Each node type must provide methods to perform seventeen different actions. This is, of course, poor man's object oriented programming. One day in the grand and glorious future Interlisp-D will metamorphose into something which properly supports object oriented programming, whereupon SEdit will be rewritten. In the meantime, this works well enough.

The methods for node types fall into four groups:

- those used to generate the presentation of a node (2 methods)
- those used to place selections and points (6 methods)
- those used to effect editing commands (5 methods)
- those which perform various housekeeping functions (4 methods)

We'll consider each of these groups in turn, specifying in detail how the methods are invoked and what they are expected to do.

Presentations and the Linear Form

The visual presentation of the data structures is represented in SEdit by a structure called the linear form. The linear form is a sequence of presentation commands which produce the desired presentation. Presentation commands are very simple, and there are only four of them:

- insert a string of characters in a given font
- insert horizontal space
- insert a given bitmap
- start a new line, with a given indentation and vertical separation from the previous line

The `Linearize` method for a node type must construct the sequence of presentation commands for a node of that type, by inserting the appropriate commands in the correct order. It may call the `Linearize` method of each of its subnodes. For instance, a very simple algorithm for linearizing lists might be:

```
output "(" in the default font
for each subnode
    if this isn't the first subnode, output some space
    linearize the subnode
output ")" in the default font
```

The actual algorithm used by SEdit is somewhat more complicated; for instance, lists usually don't fit all on one line, so it inserts line breaks at appropriate points.

To format structures such as lists properly, it's important to know how much space the presentation of each subnode will occupy. This is a problem, since the amount of space the presentation of a structure occupies often depends on the amount of space available when it is presented. SEdit deals with this by computing *width estimates* for each node. These are five values computed by each node for use by its super node:

a) if the node can be presented on a single line, the width of that presentation

b) the width of the most readable presentation of this node (its preferred presentation)

c) the width of the narrowest possible presentation of this node

d) the length of the last line of the preferred presentation of this node

e) the length of the last line of the narrowest possible presentation of this node

These width values allow the super node to make reasonable decisions on indentation and line breaks. The last line lengths are important because the super node may append additional material to the last line of a subnode's presentation, and would like to know how long the resulting line would be. These five values are stored as extra fields in each node, and referred to as the `InlineWidth` (NIL if the node can't be presented inline), `PreferredWidth`, `MinWidth`, `PreferredLLength`, and `MinLLength`, respectively.

In much of the program these width estimates are called *format values*. I've started calling them width estimates as a reminder that they are not required to be accurate, but should be quick to compute. It is important that a node's width estimates not indicate that it can be presented inline when it actually requires several lines, but other width values can be incorrect without breaking the program. On the other hand, incorrect width estimates will often lead to less than optimal presentations. The width estimates computed by the Interlisp code definitions are always correct (at least, that was the author's intention).

Each node type defines a method called `ComputeFormatValues` which will fill in the width estimates of a given node. To do so it may examine the structure the node represents and the format values of its subnodes. For instance, the ComputeFormatValues method for type list looks approximately like this:

```
InlineWidth is
    if all of the subnodes can be presented inline
        the sum of the InlineWidths of the subnodes,
            plus the width of two parentheses,
            plus the width of n-1 blanks
    else
        NIL
PreferredLLength is
    the PreferredLLength of the last subnode,
        plus the preferred indentation,
        plus the width of one parenthesis
MinLLength is
    the MinLLength of the last subnode,
        plus the width of two parentheses
PreferredWidth is
    the largest of
        the PreferredWidth of the first subnode,
            plus the width of one parenthesis
        the maximum PreferredWidth of the other subnodes,
            plus the preferred indentation
        the PreferredLLength of this node
MinWidth is
    the largest of
        the maximum MinWidth of any subnode,
            plus the width of one parenthesis
        the MinLLength of this node
```

Note that these calculations assume the node has at least two subnodes — special case rules are needed for fewer. Also, the formatting rules assumed are that

- the inline presentation of a list is an left parenthesis, followed by the subnodes separated by blanks, followed by a right parenthesis; all of the subnodes must present inline
- the preferred presentation of a list is an left parenthesis, followed by the subnodes, where each subnode after the first is on a new line indented by the preferred indentation; the final subnodes is followed by a right parenthesis
- the minimum presentation is similar to the preferred presentation, except that subnodes are indented only by the width of the left parenthesis

(Indentations are always non-negative, and specified relative to the horizontal position of the start of the node's presentation; thus the presentation of a node always occupies the quadrant to the right and below the point at which its presentation starts.)

The linearization method is given two arguments in addition to the node to be linearized and the usual context information. The first is a right margin, which it should try to keep its presentation within (if possible). This value is actually stored as another field in the node. The second argument is an index. In some situations, SEdit may request that the linear form of a node be recomputed starting part way through. In this case, the index given will be the index of a subnode, and the method should output just that part of the node's linear form which follows the presentation of that subnode.

Linearization methods are permitted to call five lisp functions to create the node's linear form:

(**\\output.string** `context string prin2? font`)
Insert `string` in the specified font at this point in the linear form. `context` is the usual context encapsulation. `string` may be any Interlisp object; its standard printed representation will be used. if `prin2?` is true, the PRIN2 representation of `string` will be used instead.

(**\\output.space** `context width`)
Insert a horizontal space of the specified width at this point in the linear form.

(**\\output.bitmap** `context bitmap`)
Insert a bitmap at this point in the linear form. It will be aligned with the line's baseline.

(**\\output.cr context** `indentation lineskip`)
Start a new line, with the specified indentation (relative to the start of this node's presentation) and separation from the previous line.

(**\\linearize** `subnode context right.margin`)
Insert the linear form of a subnode at this point in the linear form. Its linear form should not extend beyond `right.margin` (if `right.margin` is not specified, it will default to this node's right margin).

Pointing and Selecting

When the user uses the mouse to place the caret point and/or select part of the edited structure, SEdit is faced with the task of mapping the mouse's (x,y) coordinates to the appropriate structure description. These descriptions take the form of datatypes called `EditPoints` and `EditSelections`. An `EditPoint` records

- the node which owns this point (i.e. the one which will be informed if something is inserted here)
- an index, which the owning node may use to record arbitrary information about the point's location
- a type, one of Structure, Atom, or String, indicating how characters typed here will be interpreted
- a line (in the linear form) and x offset within that line, indicating where the caret should be positioned in the window if this point is displayed

`EditSelections` are similar, except that they have two indices (since selections may cover a sequence of substructures), and describe two positions (bounding the part of the linear form which should be underlined to display this selection).

The responsibility for translating mouse positions to points and selections is shared between the kernel and the type methods. The kernel determines which part of the linear form is being pointed at, and then asks the node which produced that part of the linear form to determine the point or selection. Sometimes the node will decide that in fact its super node or one of its subnodes should really be responsible, and if so it may pass on the request to them. For instance, if the user tries to insert characters at the beginning of an atom, they may actually point to the space between that atom and a preceding structure. The node which output that space was the atom's super node, i.e. the enclosing list. When it receives a request to position a character point in the space between two subnodes, it should realize that this was likely an attempt to actually edit one of the subnodes, and pass the opportunity to them.

The first method in this group is `SetPoint`. It is called with (among other things) the context (all methods are passed the edit context; from now on we'll stop mentioning it), the node, the index of the linear form item in which the mouse is positioned, the x offset within that item, and the type of point requested (i.e. the choice of mouse button used to place the point). It must fill in the fields of the point, or call some other node's `SetPoint` method to do it. It can call \\set.point.nowhere to indicate that no point is near this mouse position, and thus the point returned should not allow input.

There are two other calling sequences a `SetPoint` method may be invoked with, to allow `SetPoint` requests to be readily passed between nodes. A subnode may pass a `SetPoint` request to its super node, indicating that the point is to be placed either immediately before or immediately after itself. For instance, this is what atomic structures do when asked to insert structure. To do this, the subnode calls \\punt.set.point, passing a flag to indicate whether the point should be before or after this node. Conversely, a node may pass the `SetPoint` request to one of its subnodes, indicating that the point is to be placed at the beginning or end of that subnode (as in the example with left-clicking the space in a list, above). The `SetPoint` method can determine which case it is being asked to handle by examining its arguments.

`SetPoint` methods usually calculate the position of the point when they set it. If the linear form changes, or the point is set by some other means, it becomes necessary to recompute the position of the point so that the caret may be displayed. The `ComputePointPosition` method of the node owning the point is responsible for filling in the line and x offset values on request.

Similarly, the `SetSelection` method and `ComputeSelectionPosition` methods determine the current selection, given the mouse position. These are very similar to the corresponding methods for points, and may call \\set.selection.nowhere or \\punt.set.selection. \\set.selection.me is a useful default `SetSelection` method; it simply sets the current selection to this node.

There are two additional methods related to selections. `GrowSelection` is called when the user uses multiple mouse clicks to select structures. If the mouse handler detects a multi-click sequence, it calls the `SetSelection` method for the first click, and `GrowSelection` for each subsequent click. The `GrowSelection` method of the owner of the current selection is responsible for enlarging the selection to include the next enclosing level of structure. \\grow.selection.default is the `GrowSelection` method for most types; it simply calls \\punt.set.selection, causing the super node to become the new selection owner.

The `SelectSegment` method handles right-button mouse actions, which extend the current selection to include the item pointed to. If the mouse is pointing at part of the linear form of the owner of the current selection, the `SelectSegment` method of that node will be called to fix up the selection. Otherwise, the deepest common super node of the node selected and the node pointed to will be asked to determine the new selection, given which of its subnodes the selection and mouse are in.

Editing Operations

When an editing operation (such as the deletion or insertion of material) is performed, an appropriate method is called for the affected node, and this method is expected to fix up both the tree and the actual structure being edited. These operations are all defined in terms of points and selections, and the method invoked is that of the owner of the point or selection.

The `Insert` method takes a previously created point owned by this node and either a string of characters or a list of nodes which are to become subnodes. The appropriate changes are to be made to the tree and structure, and the point should either be adjusted to be after the inserted material or cancelled, as appropriate. Some appropriate functions to call are:

(**\\note.change** node context)
This node has changed in some way which affects its presentation. Its width estimates should be recomputed, and the linear form update appropriately.

(**subnode.changed** node context)

If the structure resulting from an editing operation is no longer EQ to the original, \\subnode.changed will inform the node's super node that it must make the appropriate updates. For instance, if a character is inserted into a litatom it actually becomes a new litatom. No change actually takes place in either atom, but the structure which contained the original atom must change to refer to the new one.

The Delete method takes a previously created selection and deletes the selected material. It returns a flag indicating whether in fact the material can be deleted; many data structures do not allow material to be deleted. It may also be asked to set the caret point so that inserted material will replace that which has just been deleted.

The Replace method replaces the selected material with either a string of characters or a list of nodes (depending on the type of selection). \\replace.default is a default implementation of this method, which does a deletion followed by an insertion. This is satisfactory for many data structures, but fails for those which do not allow deletions (e.g. quoted structures).

The Split method is only required for those types which allow character points (i.e. atoms and strings). When a delimiter (e.g. blank or cr for atoms, double quote for strings) is inserted at such a point, the node's Split method will be called. The usual behavior is to change the caret point to a structure point, and to separate the structure into two if the point was not at the beginning or end of the structure.

The BackSpace method implements the action of the backspace key. Given the caret point, the method is expected to make the appropriate deletion and adjust the point accordingly. Sometimes only the latter is necessary; for instance, if the point is positioned immediately after a list, backspace merely moves the point so that it is after the last element of the list (this corresponds to the action of the right parenthesis moving the point to after the parenthesis).

Miscellaneous Methods

There are four other methods for node types to implement, which are invoked by SEdit when the effects of some editing command might be important to a node other than that directly affected. The first of these, SubNodeChanged, was mentioned above; when the structure associated with a subnode is replaced with one which is not EQ to the original one,the super node's SubNodeChanged method will be invoked so that it can fix up its structure appropriately.

Two other methods are used to implement copying and moving structures. When a move or copy selection is made, the CopySelection method of the node owning the selection will be invoked, and passed the selection, a flag indicating what type of selection was made, and a description of the destination. The destination will be either the context of this or another SEdit process, or NIL, indicating that the copy or move is being made to a non-SEdit process, so the material should just be BKSYSBUFed. The default method \\copy.selection.default provides an implementation of this method suitable for most applications. If this is used, the CopyStructure method must be defined. CopyStructure is called with a node which has been constructed as a copy of an existing node. Because the copy operation should create new structure rather than just creating another pointer to the same structure, the CopyStructure method must fill in the Structure field of the node with the appropriate newly created structure. The Structure fields of its subnodes will already be copies of their structures, so all that is usually required is to create a new data structure out of these.

When material is moved or copied to other parts of the structure, the position at which it is inserted may imply a different parsing of the structure than that from which it is taken, since the parsing is context dependent (remember the parse mode?). In such a case, the new super node may ask the node to reparse itself, using the ReParse method. This usually involves minor adjustments to presentation, although in the worst case it may involve completely parsing the structure again from scratch.

Editing Interlisp Code with SEdit

The Interlisp editing definitions configure SEdit as an editor for programs written in Interlisp-D, and are ultimately intended as a replacement for DEdit, the system display editor. Although the current system is still missing many convenience features, it currently provides a workable alternative to DEdit. This document provides detailed information on using SEdit as a code editor. It is assumed that the reader has read the introduction to SEdit, and is familiar with the Interlisp-D programming environment. This part of SEdit is under active development; this document will be changed as improvements are made.

Running SEdit

After loading SEdit, the function `SEdit` allows it to be installed and de-installed as the default system display editor. Executing `(EDITMODE 'SEDIT)` will cause future edit requests (from functions such as `DF`, from Masterscope, and from inspectors and browsers) to use SEdit instead of DEdit; executing `(EDITMODE 'DEDIT)` will revert to using DEdit. The function will return the previous editor state (`SEDIT` or `DEDIT`).

Unlike DEdit, SEdit does not run in the process which invokes it. This has some important effects:

a) SEdit processes can be started and stopped in any order. The windows may be shrunken and kept around indefinitely if desired.

b) Calls to editing functions such as `DF` return as soon as the process is started, rather than waiting for the editing to be completed (however, when SEdit is invoked from Masterscope, it forces Masterscope to wait until the user indicates that they are done editing (by closing or shrinking the window); this is simply a convenience to avoid `Edit where any` commands from immediately covering the screen in hundreds of edit windows).

c) As a consequence of (b), some functions normally performed by `DF` (such as informing the file package that the function has been changed and needs to be saved, unsaving the definition of a compiled function, or updating the "last edited" date) are instead performed by SEdit, and often at different times (since SEdit can't wait until the user is "done editing").

Commands

At present, SEdit has no attached menu of commands. Many of the commands in DEdit's menu (such as `Before`, `After` and `Replace`) are completely unnecessary in SEdit (because of its more uniform interface). Some of them (such as `Delete`, and soon `()in` and `()out`) are provided by keyboard commands. A menu may be added in the future with the introduction of more obscure commands.

Pointing and Selecting

Like TEdit, SEdit maintains a current insertion point at which typed, copied, or moved material will be inserted. The point is set by moving the mouse to the desired position and clicking a mouse button, and is indicated by a flashing caret. Unlike TEdit, SEdit has two types of points: structure points and character points. Structure points are allowed within non-atomic structures which have a variable number of components (i.e. lists); they indicate that another Lisp structure can be inserted at the indicated position. In normal (NIL-terminated) lists, structure points can be placed before the first element, after the last element, or between any two adjacent elements; in a non-NIL terminated (dotted) list, points may not be placed anywhere after the dot. Character points are positions at which individual characters may be inserted (rather than whole Lisp structures), and are allowed in atoms and strings. The caret changes to reflect the type of point: structure points look like '▲', and character points look like '∧'.

Similarly, both structures and individual characters can be selected. A selection may be

- one of the characters in the pname of an atom or string

- a sequence of consecutive characters in the pname of an atom or string
- a lisp structure presented as an entity (e.g. an entire list, string, atom, quoted structure, etc.)
- a sequence of such structures appearing consecutively in a list

Note that not all lisp structures are presented as distinct entities, and so not all will be selectable. For instance, the individual cons cells comprising a list are usually not separately selectable. Also, extra characters added to the presentation as punctuation are not individually selectable; you can't select the left parenthesis of a list, or the closing quotation mark of a string.

The type of selection and point made depends on the mouse button used. The left button selects characters and places character points; the middle button selects structures and places structure points (this is supposed to be reminiscent of TEdit). The right button extends the current selection to the smallest selection which covers the current mouse position. As an added convenience, clicking with the left or middle button more than once in the same spot will enlarge the current selection one step, through enclosing layers of structure.

Inserting and Replacing

To insert characters in an atom or string, use a mouse button to place a character point at the desired location and just type the characters. As each character is typed, it will be inserted and the caret point will be moved after it. The `Backspace` key deletes the character to the left of the caret.

To insert new structures in lists, place a structure point at the desired location and type one of

- a left parenthesis to insert a new list
- a double quotation mark (`"`) to insert a new string
- a normal character (i.e. one with syntax class OTHER) to insert an atom beginning with that character

In the first case, an empty list will be inserted and the caret will be moved inside it. In the second, an empty string will be inserted and the caret will become a character point inside the string. In the third case, a new atom will be inserted, and the caret will become a character point to allow appending more characters to the atom's name.

There are a few other characters which are recognized specially:

- a right parenthesis places the caret point immediately after the list immediately enclosing it
- a double quotation mark, while inserting characters in a string, places the caret point immediately after the string (if it was after the last character in the string) or splits the string into two strings (if it was between two characters)
- a blank or carriage return, while inserting characters in an atom, places the caret point immediately after the atom (if it was after the last character) or splits the atom into two atoms (if it was between two characters)

These characters all leave the caret point ready to read another structure. The rules may sound a little bizarre, but they work out to give just the right behavior — typing in the printed representation of a lisp structure will give you that structure. (At present, this only works for (undotted) lists, string, litatoms, and numbers; soon dotted lists and quoted structures will also be implemented).

Special characters, such as parentheses, spaces, and double quotation marks, can be inserted in atoms by preceding them with the escape character (a percent sign).

To replace structure, it is selected "pending delete". Pending deletion selections are made whenever the current selection is extended using the right mouse button (as in TEdit). To distinguish them from normal selections they are displayed by outlining the selected material, rather than underlining it. When structures or characters have been selected pending delete, typing anything will cause them to be replaced with the new material.

Copying, Moving, and Deleting

Structures or characters may be copied or moved from one part of an SEdited structure to another, between two SEdited structures, or between an SEdited structure and any other Interlisp process which will accept or produce character string representations of lisp structures. To copy material, place a point of the appropriate type at the desired destination, and then select the desired material while depressing the `Copy` key (`Shift` on the Dorado keyboard). Selections made with the `Copy` key depressed will be displayed by a gray underline. As soon as the `Copy` key is released, a copy of the current selection is inserted at the point. If the TTY process is a non-SEdit process, a printed representation of the selected material will be BKSYSBUFed for it to read.

Moving material is done in a similar fashion,except that the `Move` key is depressed while making the selection. Move selections are displayed by a gray outline. As soon as the `Move` key is released, the selected material will be inserted at the current caret point and deleted from its original position. On the Dorado keyboard, Move selections are indicated by depressing both the `Shift` and `Control` keys.

Material may also be deleted in this fashion. Depressing the `Control` key while making a selection will cause the selected material to be deleted as soon as the `Control` key is released. Delete selections are displayed by inverting the selected material (displaying it white-on-black instead of black-on-white).

Whenever selections are being made, the selection is considered complete only when the mouse buttons *and* any modifier keys (`Copy`, `Move`, etc.) have been released. Thus, selections requiring more than one mouse click (e.g. sequence selections) can be made by keeping the modifier key depressed throughout the process. Alternatively, if the wrong modifier key is initially depressed, it can be released and another depressed as long as a mouse button is held down during this time. To completely abort the selection, click a mouse button outside the window before releasing the modifier key.

There are two other ways of deleting material. The `Backspace` key, as was previously mentioned, deletes the character to the left of the caret (this strictly true only when the caret is a character point in an atom or string; at other times it does other, reasonable things — you'll have to try it out to find out exactly what). The `Delete` key deletes the current selection. (Note that this is different from the `Control` key, which is a selection modifier; with `Delete` a normal selection is made and then the `Delete` key is depressed, while the `Control` key is depressed while the selection is actually being made — the choice of which is to use is a matter of personal taste.)

Formatting

SEdit attempts to display the structure being edited in as readable a fashion as possible, while keeping within the width of the display window. It uses fairly conventional rules for pretty-printing lisp, augmented with some special formatting rules for Interlisp special forms (e.g. LAMBDA expressions and CLisp). The components of these special forms are given indentation based on their function within the form, and special keywords are displayed in bold face to improve readability. Some effort is made to propagate the width constraint information so that relatively uniform indentation is used, rather than having complex nested expressions end up mashed against the right edge of the window. In extreme cases SEdit will extend the presentation past the right edge of the window rather than produce too ugly a presentation. If this happens, a horizontal scroll bar will be added to the window to allow editing the whole presentation.

SEdit Windows

The windows SEdit creates behave like all good Interlisp windows. They may be moved, reshaped, and scrolled. If they are reshaped to a different width, the formatting is recomputed to make the best use of the available space. The may be shrunk, producing a relatively unexciting icon adorned with the name of the variable or function being edited. When they are shrunk, the process reading commands from the keyboard is deleted (to save stack space, and allow keeping a large number of SEdit windows around), but it is automatically recreated as soon as the window is expanded again, so this should be effectively invisible. Closing an SEdit window or icon terminates the editing and releases the data structures used.

Unlike DEdit, it is not clear when to consider an SEdit editing session complete. The user may start an edit process, do some editing, shrink the window, expand it again later, etc. For some purposes it is important to have such a notion. For instance, the file package must be informed when a function is edited, so that the new definition can be saved to the appropriate file. If a function has been compiled, and the source is then edited, the compiled code should be discarded. If the editor is invoked on a sequence of structures by Masterscope or EDITFNS, it must know when it is time to go on to the next structure. When a function has been edited, a new comment should be added recording the time and date and the initials of the programmer. All of these cases require some way of indicating that this set of editing operations is completed, even if the editing process is not to be terminated. At present, SEdit handles this by assuming that editing is complete when the window is closed or shrunk.

Confusing SEdit

SEdit currently assumes that no changes will be made to the structure being edited during an edit session, other than those made by SEdit itself. Of course, since SEdit exists in a lisp environment replete with shared structures and destructive functions, there is no way to enforce this. For instance, while editing a variable whose value is a list you could (from an executive window) run a function to destructively change that list. There is no way for SEdit to notice the change as it happens, and at present it will not recover gracefully if it encounters the inconsistency later on.

To avoid causing these problems to itself, SEdit automatically avoids starting two edit processes on the same variable or function. This is only a partial solution, however, and the user is advised to watch out for this problem. If you suspect that a structure being edited has been changed by some other process, simply close the window and start another edit process. This will force SEdit to reexamine the structure.

In the immediate future SEdit will be fixed to detect the most common case of such changes, namely corrections made by DWIM. It is also planned to make SEdit much more robust in the face of other changes it may detect (in the not quite as immediate future).

How SEdit Works

SEdit incorporates a variety of complex algorithms and data structures. Although the code is commented, there is no logical place within it to properly describe the use of these. Therefore this document provides an overview of the most interesting and tricky parts of the program. If you want to understand exactly what the code does, this isn't a replacement for actually reading it. This is, however, a highly recommended introduction to the code; it's probably not worth trying to understand the code if you haven't read this first.

The Code

The code for SEdit currently resides in five files, named SEdit, TopLevel, SEditWindow, Linear, and IntrLsp. The first four of these comprise the SEdit kernel and its interface to the rest of Interlisp-D; the last is the definitions which configure SEdit as an editor for Interlisp code. The approximate division of labor is:

SEdit
process initialization, keyboard command loop, top-level method invocation, building and manipulating the edit tree

TopLevel
interface to Interlisp-D editing functions and file system, starting and managing SEdit processes

SEditWindow
window system interface, mouse selection and pointing, scrolling

Linear
building and manipulating the linear form, optimized screen updating

IntrLsp
methods for the standard Interlisp types

Included with the code listings is a directory of SEdit function names, identifying the file within which they occur. Within each file listing the functions are sorted into alphabetical order. SEdit currently uses the convention of preceding its function names by a pair of backslashs, to clearly distinguish them from other parts of the Interlisp system. At some point a more conventional prefix will be chosen.

Control Flow

SEdit sessions are started via the system function EDITL, which originally invoked the TTY structure editor. When SEdit is enabled, EDITL is modified to call SEdit. SEdit first attempts to find an active SEdit already editing the same function or variable. If one is found, SEdit makes sure that the edit window is expanded and not buried on the screen, and then returns. In this way it avoids starting several SEdit processes on the same structure, which could lead to major confusion. If no active SEdit is found, it will start a new one. It places a window, either by asking the user where the window should be or by using the window position of a previous SEdit (to avoid prompting the user over and over again). Once it has a window, SEdit starts a new process running the keyboard processing loop, and waits for it to signal that it has initialized.

A further complication is that EDITL may be invoked with a sequence of TTY editor commands which are to be performed. Rather than attempt to implement the large and baroque command set of the TTY editor, SEdit simply calls the TTY editor to execute the command sequence, but with the constraint that should the command sequence involve a pause for user editing (i.e. the TTY: command), the TTY editor will call SEdit back. This mechanism is used heavily by Masterscope, which usually passes a sequence of editor commands which search for particular parts of the function and then allow the user to edit them.

If EDITL is invoked without any commands, it returns as soon as the command loop signals that initialization is under way. The editing will be handled by the new process, and whoever invoked SEdit can now go on with other things. Alternatively, if a command sequence is given, SEdit will wait until the command process signals that the user has indicated editing is complete before returning. This is quite important; if EDITL doesn't wait under these conditions programs like Masterscope may try to start dozens of SEdit procedures, without giving the user time to actually do any editing.

The command loop which is executed by the SEdit process (\\sedit) first checks to see if this is a new SEdit context or an old one. If it's new, the structure must be parsed to generate the edit tree and linear form, window parameters must be initialized, and the initial presentation must be displayed. If this is a continuation of an old edit session, any pending adjustments to the presentation are completed, and the current selection is redisplayed. Either way, \\sedit then enters a loop reading single characters and executing the appropriate action.

Keyboard Input

The interpretation of characters typed from the keyboard is determined by an Interlisp readtable, which maps characters to syntax classes. The syntax classes understood by SEdit are:

STRINGDELIM: this is the delimiter for strings (i.e. double quote)

SEPRCHAR: white space, e.g. blank and carriage return

ESCAPE: when preceded by this character, other characters should be treated as syntax OTHER (the usual escape character is %)

OTHER: a "normal character", suitable for inclusion in the pname of a litatom

(*type where when function*): a read macro. *function* is the function to be invoked when this character is typed. *type* determines what information will be passed to the function, and how its result will be interpreted; at present the only legal value is INFIX, indicating that the function will be passed the complete edit context, and is free to make any changes whatsoever. *where* and *when* control under what conditions the function will be invoked; the valid combinations are:

(ALWAYS IMMEDIATE)
the function will be invoked whenever the character is input (e.g. Delete)

(ALWAYS NONIMMEDIATE)
the function be invoked except when the character is typed as part of a string (e.g. left parenthesis)

(FIRST NONIMMEDIATE)
the function will only be invoked when this character is typed at a structure point, i.e. as an atom on its own (e.g. period)

Note that these are not exactly the conventional interpretation of readtable entries, but have been adapted to SEdit's needs.

The action produced by a character depends on both the syntax class of the character and the type of the current point (string, atom, or structure). For instance, a SEPRCHAR typed to a string point will insert that character in the string, typed to an atom point will split the atom, and typed to a structure point will simply be ignored. An OTHER character typed to a string or atom point will be inserted, but typed to a structure point will cause a new node, representing the single character atom, to be created and inserted.

Mouse Actions

In the Interlisp-D window system the effects of mouse movement and buttons are determined by a special mouse process, which invokes functions attached to the windows. The effect of this is that activities like selecting and pointing are actually run under the mouse process, rather than under the SEdit process. This has several ramifications. First, the mouse process must be able to easily access the state of the editing process, so that it has the necessary information available to carry out the actions. For this and other reasons the entire state of the SEdit process is encapsulated in a single data structure, called an `EditContext`, which is then attached as a property of the window.

Second, access to the editor's data must be controlled, so that multiple processes don't interfere with each other. This is achieved through a monitor lock on the `EditContext` (called the `ContextLock`).

The Linear Form

The data structure used to encode the linear form must satisfy several constraints. First, as its name suggests, it is treated as a sequence of items. The window is repeatedly being redrawn from the linear form, and usually the segment which is redrawn cuts across the tree structure. This requires being able to start iterating through the linear form from any point, and continue until some other arbitrary point, perferrably without having to maintain a stack. On the other hand, it is generated and modified hierarchically; the kernel must be able to quickly extract and replace the linear form for any node in the tree, without affecting the nodes above or (in some cases) below it. To achieve this, the linear form for each node is stored as a list pointed to by the node, which ends with a pointer back to the node. This list may contain linear items and subnodes of the node, indicating that their linear forms are to be inserted at that point. Finally, each node contains a pointer (called the `LinearThread`) which points to the position in its supernode's linear form at which it appears. This threading allows any part of the tree structured linear form to be easily and efficiently traversed.

One complication to this scheme is that because of the reference counting garbage collector in Interlisp-D, we don't actually make the tail of the linear form point back to the node directly. This would introduce a circular chain of pointers, preventing these structures from ever being garbage collected. Instead, we use a data structure called a `WeakLink`, which contains one non-reference counted pointer field. Of course, the use of such pointers can make debugging hazardous; unfortunately, they are pretty much unavoidable in memory management scheme such as Interlisp-D's, and SEdit uses them in a number of places.

Most of the items which can appear in the linear form are relatively simple, but `LineStarts` are the exception. `LineStarts` are inserted by the linearization procedures to indicate that a new line is to be started, but are used by the kernel a lot of extra information. First, each `LineStart` is linked to the `LineStart` before and after it, allowing efficient access to sequential lines of the linear form. Second, each `LineStart` records the maximum ascent and descent of the items which appear on that line, since these determine the amount of vertical space required to determine the line. Since lines can contain arbitrary combinations of text in different fonts and sizes, as well as bitmaps of arbitrary size, each line's ascent and descent must be separately computed, and may change with any change to the nodes appearing on that line. Third, each `LineStart` has a pointer to the node in whose linear form it appears; although this could be determined by scanning along the linear form, it's used often enough that we cache it. Fourth, each `LineStart` stores the y coordinate of that line's baseline. This is a function of that y coordinate of the preceding line, the line ascents and descents, and the separation between the lines.

Thus, the algorithm for updating the window from the linear form is something like this:

```
(* pointer is the current position in the linear form)
while pointer is not NIL do
    if pointer is a list, then
        let item be the first element of the list
        if item is a number (* horizontal space), then
            increment current x by item
        elseif item is a LineStart, then
            set x to the indentation of item
            set y to the baseline of item
        elseif item is a bitmap or string item, then
            paint this item and increment x by its width
        else item is a subnode
            set pointer to the linear form of item
    else pointer is a WeakLink
        set pointer to the CDR of the linear thread of
            the node pointed to by this WeakLink
```

Incremental Relinearization

When the node structure is changed, SEdit is faced with the task of updating the screen as efficiently as possible. This is actually treated as two problems: first, the new linear form must be computed as efficiently as possible, and second, given the changes that occurred in the linear form, we wish to make the corresponding adjustments to the window, with a minimum of repainting.

The first problem is dealt with using a number of tricks. First, as changes are made to the tree no attempt is made to update the linear form immediately; all that is done is to keep a list of those nodes which have been changed and thus need their linear forms recomputed (this is the purpose of \\note.change). This list is kept sorted by depth, and duplicates are discarded Once SEdit decides that it's time to update the window, it finds a minimal set of nodes to relinearize, such that all of the changed nodes are contained directly or indirectly (as subnodes of an included node), and the width estimates of these nodes have not been changed by the editing (hence the linear form of their super nodes won't have changed). The linear form of each of these nodes is recomputed. Second, during this relinearization, subnodes of a relinearized node will not be relinearized if their structure hasn't changed, and SEdit is able to determine that their resulting linear form won't have changed (e.g. their margins haven't changed). In practice, this typically results in a major decrease in the amount of relinearization involved.

The second problem is more difficult. The idea is that we want to reuse as much of the existing window display as possible. If part of the linear form is already displayed in the correct place, no changes should be made to it. If part of it is already displayed, but in the wrong place, SEdit tries to use a bitblt call to copy those bits to the correct location rather than reconstructing them. This is complicated by the fact that material copied may be less than one line or span several lines. The line its moved to may have greater or less ascent and descent than the one it came from; in fact, the ascent and descent of the destination line won't be known until it's completed. To deal with this, SEdit builds a second description of the line, in parallel with the construction of the linear form. This structure describes the line as a sequence of blocks, each of which represents a subsequence of the linear form. Each block may also already appear in the window, in which case the coordinates from which it can be retrieved are also recorded. At the end of each line, SEdit examines the sequence of blocks, determines which ones actually represent useable blocks of bits, copies them, and repaints any gaps. To avoid overwriting bits which it may later want to use, SEdit will sometimes shift whole parts of the screen out of the way; this introduces additional complications, since coordinate transformations are now required to determine the actual location of the desired bits.

A Word About Lines and LineStarts

SEdit defines a `LineStart` data type to record information about a line in the linear form. The `LineStart` does not record the actual linear items which appear on that line; that information is implicitly recorded in the linear form itself. The linear form is an list of linear items, and `LineStarts` are items. Thus, the items on a line are those which follow it in this list (until the next `LineStart`). Since this information is often needed, many of the places where one might expect a pointer to a `LineStart` actually contain a pointer to the linear form which start at that `LineStart`, i.e. a list, the first element of which is the `LineStart`. This is referred to as a `Line`.

Pretty Printing

The code listings at the end of this document were generated by SEdit's formatting routines. This seemed like an obvious thing to do, but a couple of caveats are in order. First, the formatting rules are still incomplete; in particular, they do a miserable job on **create** expressions (which have a very non-LISP like syntax). Second, quite little modification was required to add this capability to SEdit, but the result isn't an ideal pretty printer. It does a good job (better than Interlisp's pretty printer), and hopefully will soon do an even better job, but it's a very expensive way to do it. Constructing the complete edit tree for a one-shot linearization consumes excessive memory and time, and consequently printing large files of functions requires considerable patience.

Contexts, in detail

(an annotated description of the `Context` data structure definition)

Environment: an `EditEnv`

The Environment provides a number of parameters controlling the editing process. The idea is that different environments are built to describe different editing tasks (editing different languages, or just different edit styles), and then shared between all edit contexts of that type.

DisplayWindow: a `WINDOW`

This is the primary window, in which the edited structure is displayed.

EditType: probably a litatom

Doesn't affect the editing, but used (in conjunction with IconTitle) to help identify the source of the structure being edited. One of VARS, FNS, PROP, etc.

IconTitle: a string or litatom

The name of the structure being edited, used to title the window and icon. Also, in conjunction with EditType, used to determine whether an existing SEdit session is already editing a structure the user has asked to SEdit.

ContextLock: a `MONITORLOCK`

The monitor lock used to control access to this Context.

CompletionEvent: an `EVENT`

SEdit will signal this event when the user shrinks or closes the window, or otherwise indicates that they have done enough editing. When called under the TTY editor, the process which spawns the SEdit command loop will wait for this event.

WindowLeft, WindowRight, WindowBottom, WindowTop: integers

During screen update processes, SEdit caches information about the window dimensions here, since they're used so frequently.

CurrentX: an integer
CurrentLine: a `Line`

While generating the linear form, SEdit uses these to record the start of the current line and the horizontal position within that line.

LinearPointer: a position within the linear form

This is the current position within the linear form, for purposes of comparison and insertion.

LinearPrev: a position within the linear form

This is one step behind the LinearPointer, to allow fixing up pointers when an item is inserted. If LinearPointer points to the first item within the linear form of a node, LinearPrev will point to that node.

Root: an `EditNode`

Points to the root node in the edit tree.

LastLinearizedSubNodeIndex: an integer

During linearization, this field is used to record the last subnode linearized of the current node (as a consistency check).

ChangedNodes: a list of `EditNodes`

This list records which nodes in the tree have had their structure changed and hence require relinearization. It's headed with a dummy item (NIL) to simplify insertion, and sorted by decreasing depth. Most of the time, this list should contain exactly those items whose Changed? field is T.

CaretPoint: an `EditPoint`

Records the current insertion point.

Selection: an `EditSelection`

Records the current selection.

Caret: a `CURSOR`

This is the mark to be displayed at the current caret point (hollow or solid, depending on the type of point).

SelectionDisplayed?: a boolean

Records whether the current selection has been underlined, outlined, or whatever.

LastMouseX, LastMouseY: integers
LastMouseType: one of (`Atom`, `Structure`)

Records the position of the last mouse click, and the button used, to detect multi-click sequences.

\X, \T: integers

These slots in the `EditContext` are used at one point in the program to return multiple values from a function; too bad there's no better way.

FirstBlock: a `LineBlock`

The first in the sequence of blocks constructed to describe the current line.

CurrentBlock: a `LineBlock`

The last (so far) in the sequence of blocks being constructed to describe the current line.

Matching?, Below?, Visible?: booleans

These are used while constructing the block sequence to keep track of our current state.

RepaintStart: a position within the linear form
RepaintLine: a `LineStart`
RepaintX: an integer

Also used to keep track of state while constructing the block sequence.

ShiftY, ShiftDown, ShiftRight: integers

When the block shifter moves the rest of the window contents out of the way, these variables record what has been done to allow translating coordinates. Everything below ShiftY has been shifted down by ShiftDown; everything on the current line past the current position has been shifted right by ShiftRight.

RelinearizationTimeStamp: an integer

During relinearization, the position of a line may move. However, the old position of that line may be later needed to locate useful bits. When the position or dimensions of a possibly useful line are changed, the old values are cached. Since the values are only good for the current relinearization, they are marked with this time stamp, and each relinearization its value is increased.

Environments, in detail

(an annotated description of the `Environment` data structure definition)

ParseInfo: a `PLIST`

This `PLIST` maps TYPENAMEs of edited data structures to the functions which can parse them.

ParseInfoUnknown: a function name

This is the function called to parse any data structure whose TYPENAME doesn't appear in ParseInfo.

DefaultFont, ItalicFont, KeywordFont: font descriptors

These are the fonts to be used when formatting.

DefaultLineSkip: an integer

If the linearization procedure doesn't specify the vertical spacing between lines this value will be used.

ReadTable: a `READTABLE`
This is the table used to determine the syntax of keyboard input and the interpretation of command characters.

SpaceWidth: an integer
This is the default space to leave between adjacent items in lists, etc.

DefaultIndent, MinIndent, MaxIndent, MaxWidth: integers
These values control the formatting of list structures.

LParenString, RParenString, DotString, QuoteString: `StringItems`
To prevent repeatedly constructing `StringItems` for the standard punctuation symbols, they're cached in the environment and shared. These must be reconstructed if the fonts are changed.

EditNodes, in detail

(an annotated description of the `EditNode` data structure definition)

NodeType: an `EditNodeType`
The type of this node, providing the set of methods.

ParseMode: a litatom
The parse mode in which this node was parsed.

SuperNode: an `EditNode`
This node's super node; NIL if this is the root.

Depth: an integer
The depth of this node within the tree; the root has depth 0.

SelfLink: a `WeakLink`
To avoid building uncollectable circular structures, the linear form ends with `WeakLink` back to the node. To avoid consing `WeakLinks`, we cons one and cache it here.

SubNodeIndex: an integer
This is the index of this node within its super node's subnodes.

Structure: anything
This is the structure this node actually represents.

Changed?: a boolean
True if this node has been changed and will require relinearization. Most of the time, true iff this node is on the context's list of changed nodes.

InlineWidth, PreferredWidth, PreferredLLength, MinWidth, MinLLength: integers
The width estimates computed by this node's ComputeFormatValues method.

SubNodes: a list of `EditNodes`
The subnodes of this node.

LinearForm: a list of linear items
The linear form of this node, terminating in a `WeakLink` back to the node.

LinearThread: a list of linear items
The position of this node within its super node's linear form (hence a list whose CAR is this node)

Unassigned: anything
Available for use by this node's methods to cache any additional information they wish to record.

StartX: an integer
The horizontal position at which the linear form of this node begins.

RightMargin: an integer
The right margin used when formatting this node.

ActualWidth: an integer
The maximum horizontal offset of any part of the presentation of this node from the starting position.

ActualLLength: an integer
The offset of the end of the last line of this node's presentation from the node's starting position.

FirstLineLinear, LastLineLinear: `Lines`
The line on which this node's linear form begins, and the line on which it ends.

Inline?: a boolean (computed)
True iff FirstLineLinear and LastLineLinear are the same.

FirstLine, LastLine: `LineStarts` (computed)
The CARs of FirstLineLinear and LastLineLinear.

EditNodeTypes, in detail

(an annotated description of the `EditNodeType` data structure definition)

Name: a string or litatom
This is not used by SEdit, but provides a handy point of reference when debugging.

ComputeFormatValues, Linearize , ReParse , SubNodeChanged , SetPoint , ComputePointPosition , ComputeSelectionPosition , SetSelection , GrowSelection , SelectSegment , Insert , Split , Delete , Replace , CopyStructure , CopySelection , BackSpace : function names
These are the methods of this node type.

LineStarts, in detail

(an annotated description of the `LineStart` data structure definition)

NextLine, PrevLine: `Lines`
These lines immediately before and after this one in the linear form.

Node: an `EditNode`
This is the node in whose linear form this `LineStart` was generated.

LineAscent, LineDescent: integers
The maximum ascent and descent of any item on this line, and hence the dimensions of the line.

LineSkip: an integer
The vertical separation of this line from the line preceding it.

Indent: an integer
The amount by which this line is horizontally indented.

LineLength: an integer
The total length of this line (including indentation).

YCoord: an integer
The vertical position of the top of this line (including the LineSkip). Since we set up the window's coordinate system with (0,0) in the top left corner, YCoords are always negative or zero.

CachedY, CachedAscent, CachedDescent: integers
During relinearization, we may need to determine what the position and dimensions of a line were after we've changed them. When this is a possibility the old values are saved here.

CacheTime: an integer
When the old position and dimensions are cached, the current value of the context's RelinearizationTimeStamp is recorded here, so that we can quickly determine when the cached values are out of date.

LineHeight: an integer (computed)
The sum of LineSkip, LineAscent, and LineDescent.

BaseLineY: an integer (computed)
YCoord minus the sum of LineSkip and LineAscent.

NextLineY: an integer (computed)
YCoord minus LineHeight; should be equal to the YCoord of NextLine.

OldTop, OldBottom: an integer (computed)
If the cache values are up to date (comparing their time stamp with that of the context) use them; otherwise use the current values.

StringItems, in detail

(an annotated description of the `StringItem` data structure definition)

String: a litatom or string
The text to be displayed.

Font: a font descriptor
The font in which the text should be displayed.

PRIN2?: a boolean
True if the PRIN2-name of String should be used, rather than the pname (i.e. string delimiters will be displayed, command characters will be escaped).

Width: an integer
The width of this item when displayed.

EditPoints, in detail

(an annotated description of the `EditPoint` data structure definition)

PointNode: an `EditNode`
The owner of this point, i.e. the node in which insertion will take place.

PointIndex: an integer
Used by the PointNode to record the point's position within the node.

PointType: one of (`Structure`, `Atom`, `String`)
Determines how characters typed at this point are to be interpreted.

PointLine: a `LineStart`
PointX: an integer
The position in the window at which the flashing caret should be displayed while waiting for keyboard input.

EditSelections, in detail

(an annotated description of the `EditSelection` data structure definition)

SelectNode: an `EditSelection`
The owner of this selection.

SelectStart, SelectEnd: integers
Used by the SelectNode to record the selection's boundaries within the node.

SelectType: one of (`Structure`, `Atom`, `String`)
If typed input replaces this selection, this determines how it will be interpreted.

SelectStartLine, SelectEndLine: `LineStarts`
SelectStartX, SelectEndX: integers
The boundaries of the area to be highlighted when this selection is displayed.

DeleteOK?, ReplaceOK?: booleans
The use of these flags is not completely implemented yet.

LineBlocks, in detail

(an annotated description of the `LineBlock` data structure definition)

BlockStart: a position within the linear form
The start of the segment of linear form this block represents (the end is determined by the start of the next block).

BlockNewX: an integer
The horizontal position at which this block will be displayed.

BlockWidth: an integer
The width of this block when displayed.

NextBlock: a `LineBlock`
The next block in the sequence.

Bits?: a boolean
True if the segment of linear form represented by this block is already displayed on the screen.

BlockX, BlockBaseLine, BlockAscent, BlockDescent: integers
If Bits? is true, these fields give the current position of the block's presentation.

An Extensible Structured Data Editor for Interlisp-D

Lisp environments are populated by a variety of complex linked data structures, particularly linguistic structures (such as programs written in Lisp and other languages). These are often designed with visual representations (e.g. the pretty-printed form of Lisp code). Through such features as read macros and print definitions, the data structures may be converted to or from a textual representation. In many cases, the only convenient way to edit the structure may be to edit the textual representation and then re-interpret it. This approach suffers from several difficulties:

- writing the structure out and reading it back in will generally create an entirely new structure; this may be a problem if the structure is shared or part of a larger structure
- the requirement that enough information be written out to reconstruct the data when it is read back may conflict with the desire that the presentation be convenient to understand and change
- if the text editor is to provide any assistance (syntax checking, semantic checking, etc.) it must be integrated with the Lisp environment, and continually translating structures to and from their textual representation
- no allowance is made for nontextual graphic presentations

SEdit provides a different approach. Data structures are edited directly, with visual presentations provided as a means of communication. Like WSIWYG text editors, the editing window contains a continuously updated presentation of the data structure, and editing operations are input in terms of this presentation. Presentation and editing rules are defined for each data type, and a kernel program uses these rules to perform the editing.

A set of such rules have been written to configure the editor as a tool for editing Interlisp programs. Interlisp code is pretty-printed in the window, and editing operations use a simple "point and type" user interface. This provides a more convenient way of editing Interlisp than previously existing tools, and furthermore will allow the simple construction of editors for other languages implemented in the Interlisp environment (Prolog, CommonLisp, 3-Lisp, Loops, etc.).

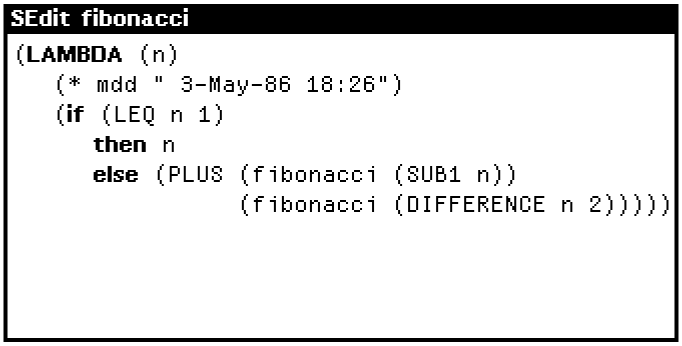
A Sample Session

Before discussing the structure and implementation of SEdit, we will try to give a feel for what it does by a brief description of some simple editing. Of course, it is a little hard to convey the flavour of as interactive a process as editing with just a few words and pictures. This discussion assumes some familiarity with the uniform editing interface which many Interlisp-D programs (and in fact most Xerox software) supports, since that was the model for SEdit's interface. The basic principles are that

- there can be a *current insertion point*, usually indicated by some sort of caret, at which inserted material appears
- there can be a *current selection*, usually indicated by underlining or other highlighting, which indicates the material to be affected by a following command (commands may change the selected material, change some aspect of it (such as font or formatting), delete it, etc.)
- the point and selection are usually placed by positioning the mouse and pressing a button; the left or middle button places the point near the cursor and selects the indicated material, while the right button extends a previously made selection to include the indicated material
- material may be selected while holding down a modifier key; instead of becoming the current selection, an action is performed as soon as the key is released. The standard modifiers are `Copy`, which inserts a copy of the selected material at a previously

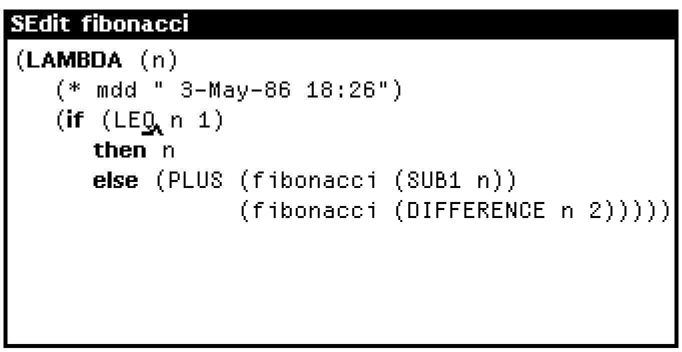
chosen insertion point; `Delete`, which deletes the selected material; and `Move`, which copies the selected material and then deletes it

To SEdit an Interlisp function after SEdit has been loaded, use the usual `(DF` *function*`)`. This creates an edit window for the function. Unlike the older display editor (DEdit), DF will return immediately, since the editing is done in a separate process. The function body will be pretty-printed in the window:

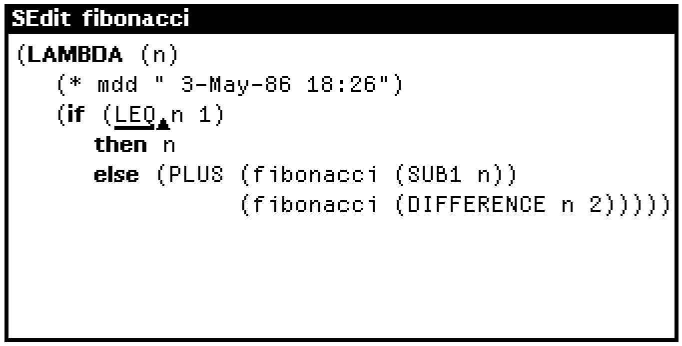


Since SEdit knows that this is Interlisp code, it uses some formatting rules specific to Lisp as well as its general rules for formatting lists, atoms, and strings. Formatting rules specify visual presentations as the positioning of text strings (in any font) and arbitrary bitmaps.

SEdit allows operations both on complete Lisp objects (such as lists and atoms) and on parts of objects, such as the individual characters in an atom's name. To do this conveniently, it uses the convention that the left mouse button is used to point to parts of structures, while the middle mouse button always points to whole Lisp structures. Thus, left-clicking the `Q` in `LEQ` selects that character, but middle-clicking there will select the whole atom (this convention matches TEdit's character/word selection convention). Furthermore, the same convention applies to insertion; a left click between the `E` and `Q` will allow inserting more characters, but a middle click will allow inserting a new structure (after the `LEQ`, since the mouse was closer to the end of the atom than the beginning). SEdit indicates the type of insertion expected by changing the caret, to '**⋀**' for a substructure and to '▲' for a structure. Thus, left-clicking the right side of the `Q` in the above window produces:



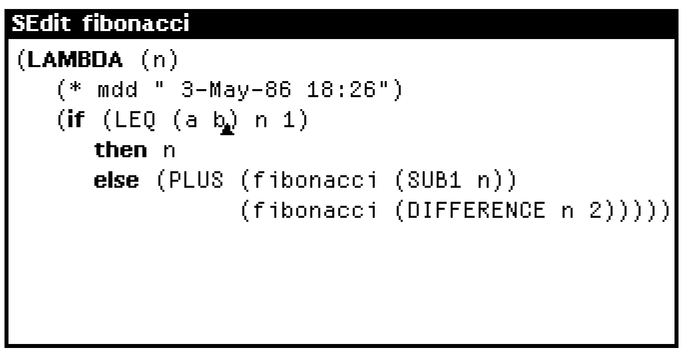
and any characters now typed would be appended to the atom `LEQ`. On the other hand, middle-clicking in the same place produces:



and if characters were inserted they would form a new atom.

Enclosing structures are selected by multiple clicks. In the previous example, if the middle button were clicked twice, the list `(LEQ ...)` would be selected; if it were clicked thrice, the `(if ...)` would be selected, etc. Sequences of structures or substructures are selected by extending the current selection with the right mouse button. If the atom `LEQ` has been selected, right-clicking the `1` would select the sequence of three structures `LEQ`, `n`, and `1` (this is *not* the same as selecting the list which contains these structures). Similarly, the `onacc` in the middle of `fibonacci` could be selected by left-clicking at one end of the sequence and right-clicking at the other end.

Some characters activate SEdit commands. For instance, typing a left parenthesis inserts an empty list at the current insertion point and positions the point inside it. Typing a right parenthesis positions the point just after the list immediately enclosing the current point. Typing a blank when inserting within an atom splits the atom at the point (unless the point is at one end of the atom) and switches to structure insertion. This allows Lisp list structures to be typed in as usual. For instance if we continue the above example by typing "`(a b `" the window will now appear like this:



(Note that the matching right parenthesis appears as soon as the left parenthesis is typed; this is an immediate consequence of having the window always be a pretty-printed presentation of the underlying list structure. Double quotes work the same way for strings.)

Modified selections allow the easy rearrangement of existing structures. For instance, to switch the order of the two arguments to `PLUS`, simply

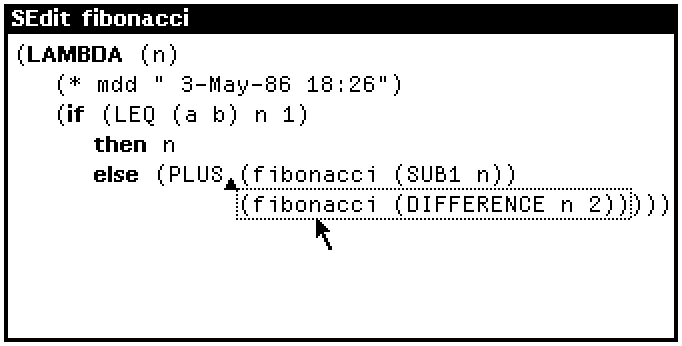
a) middle-click after `PLUS` (sets the insertion point)

b) hold down the `Move` key (we are going to move part of the structure to the current insertion point)

c) middle-click on the second `fibonacci` twice (selects the whole function call)

d) release the `Move` key (to indicate the selection is completed)

Just before the `Move` key is released, the window looks like this:



The selection is highlighted by a dotted outline to indicate that this is a `Move` selection (`Copy` selections have a dotted underline, pending-replace selections have a solid outline, and `Delete` selections are inverted). As soon as the change is made, the window is updated to show the resulting structure (SEdit determines what changes to the presentation result from the editing, and repaints as little of the presentation as possible).

An Overview of the Control Structure

The operations demonstrated above are actually performed by the cooperation of three separate pieces of code: the SEdit kernel, the SEdit user interface, and a set of methods defining the datatypes to be edited. By factoring the control this way, SEdit can be customized to edit new datatypes or provide different commands without having to modify or understand the complexities of the kernel (this is fortunate, since the kernel is quite complicated). The kernel invokes the datatype methods at appropriate times, manages the global sequencing of operations, and handles the optimization and caching necessary to update the screen efficiently. The datatype methods fall into three classes:

- methods for effecting and responding to changes in the structure being edited
- methods for positioning the point and selection appropriately
- methods for generating the visual presentation of the structure

The user interface translates mouse and keyboard events into appropriate calls on the SEdit kernel operations. It is not as modular as I would like; the keyboard event interpreter is table driven and readily extensible, but the mouse event interpreter is not extensible at present.

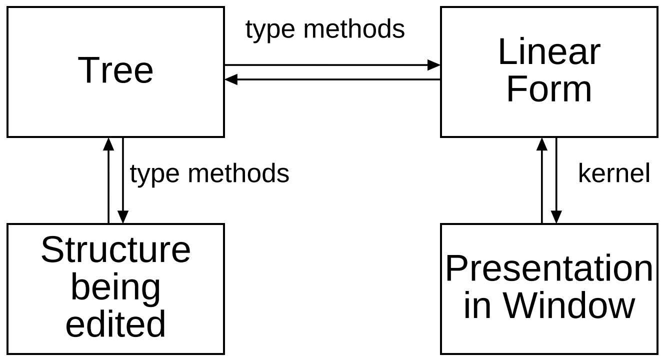
Each instance of SEdit has a *context*, which includes the structure being edited and the window in which the editing is being done. Editing operations are performed by calling kernel procedures and passing the context. The context is monitor locked, so operations may be invoked from several processes. Normally, each context has an associated keyboard process which reads keyboard commands and performs them. No editing state is maintained by the keyboard process; it can be deleted and recreated as convenient (but SEdit ensures that there is never more than one keyboard process per context).

SEdit is installed by calling the function `(EDITMODE 'SEDIT)`, whereupon it becomes the system display editor and will be used whenever the user asks to display edit a structure (through edit commands, Masterscope searches, inspectors, browsers, or whatever). The context remains active until the edit window is closed (although the keyboard process disappears while the window is shrunk).

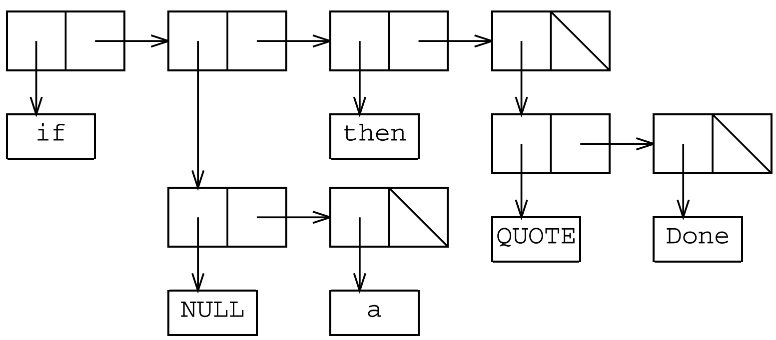
An Overview of the Data Structures

The SEdit user is aware of two data structures: the structure being edited, and the window in which it is presented. Internally, SEdit maintains two additional data structures: the *tree*, which is a representation of the structure being edited, and the *linear form*, which is a representation of the visual presentation appearing in the window. The tree provides a common representation for the underlying structures. This allows SEdit to implement a small number of simple editing operations uniformly described as tree mutations, while the type-specific methods map these into the actual changes required. It also caches various bits of information computed by SEdit. The linear form is a sequence of strings, atoms, and bitmaps to be displayed, intermixed with horizontal and vertical positioning information.

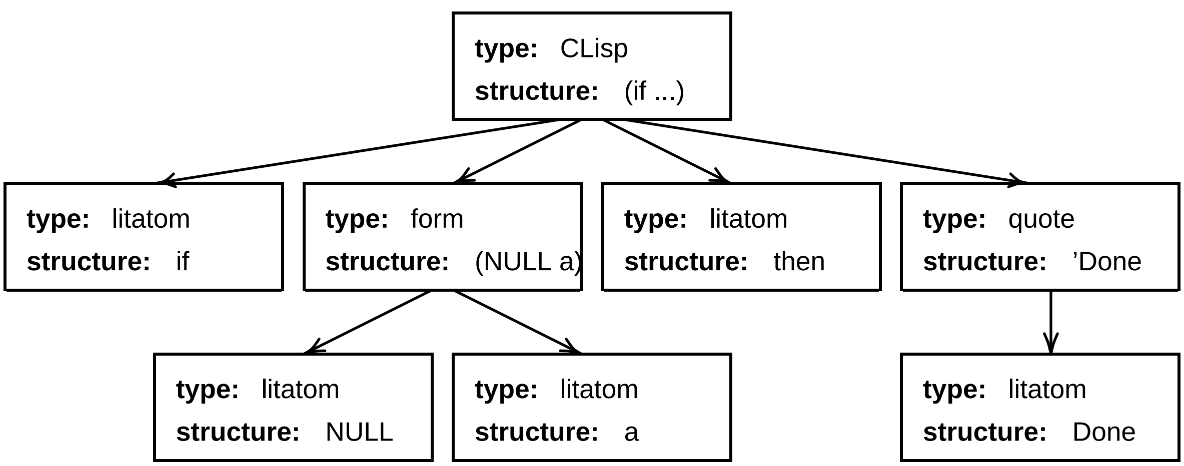
These four data structures are all representations of the same data. The constraints between them are maintained as follows:



As an example, consider editing the list structure `(if (NULL a) then (QUOTE Done))`. The underlying structure is composed of cons cells and atoms:



SEdit's tree will look like this:



Note that each node in the tree has a type, indicating the type of the structure, and a field containing the structure itself. Note also that the types need not correspond directly with Interlisp's type structure: in this example, many cons cells are grouped together as one node in the tree, and may be marked as type `list`, `CLisp` (a class of special form), `form`, `quote`, `binding`, etc. depending on the components of the structure and its position in the surrounding structure. Thus, the list `(QUOTE Done)` is classified as type `quote` above (and has one subnode), but would have been type `binding` (with two subnodes) had it appeared in a list as the second element of a form identified as a `LET` or `PROG`.

The linear form will be something like:

```
"(", default font
    "if", keyword font
skip 7 points
    "(", default font
        "NULL", default font
    skip 7 points
        "a", default font
    ")", default font
new line, indent 20 points
    "then", keyword font
skip 7 points
    "'", default font
        "Done", default font
")", default font
```

Note that although the linear form is a sequence (hence its name), it reflects the structure of the tree from which it was generated. Finally, the window will show some or all of the presentation described by the linear form:

```
(if (NULL a)
   then 'Done)
```

SEdit has two other important data structures. The context was mentioned before; it includes the data structures mentioned above, information such as the current selection and current insertion point, and all the other little bits of state SEdit needs to keep track of what is going on. The *environment* contains all of the customization information which controls SEdit's behaviour: the tables which drive the generation of the tree from the original data structure, the table for interpreting keyboard events, parameters such as default spacing and fonts, etc. At present, there is just one environment defined — the one which configures SEdit specifically for editing Interlisp code, but by creating new environments the user could have SEdit contexts for editing Prolog, CommonLoops, etc., all coexisting.

(Actually, there *is* a second environment defined, which tricks SEdit into pretty-printing Interlisp code into a TEdit document (to produce the listings later in this document), but it hardly qualifies as an *edit* environment.)

Documentation

The rest of the documentation for SEdit is in five sections:

- a more detailed account of how to use SEdit to edit Interlisp code
- a description of how SEdit is configured for editing a new type
- a detailed description of SEdit's internals
- an annotated listing of the code which implements the basic Interlisp types
- an annotated listing of SEdit's kernel

The first of these is independent of the others, but the second should be read before attempting to understand any of the following sections. If you are planning to extend SEdit to edit your own language (or change the way it currently edits one), you should also look at the third of these — a few good examples are sure to convey what the specifications missed.

Other Comments

The formatting and incremental reformatting algorithms, the incremental window update algorithms, and the various data structures were all developed by the author, so there are no references for them. Although there are several extensible prettyprinters for Lisp environments (notably Waters' GPRINT), some do not allow a sufficiently general class of presentations, most only deal with formatting lines of fixed width characters, and none deal with the (crucial) problem of incremental reformatting.

At present, SEdit is usable but far from polished. There is still room for improvement in the screen updating, and many rather desirable commands are still missing (notably *undo* and *redo*). Extra formatting rules for CommonLoops are a high priority, as is completing the interface to other Interlisp tools (e.g. the file package). Finally, the author hates the name SEdit, and promises three free features of their choice to the first person who comes up with one he likes better.

How SEdit Formats LISP Forms

If you have a list format associated with the car of a lisp form, sedit will use the information in that list format to prettyprint the form. The details of internal list format structure are documented elsewhere; this note describes an easy-to-use interface to these internal structures.

Getting and Setting Formats

Formats are associated with the cars of lists. To find out what format is associated with a particular function name, say FOO, you say (GET-FORMAT ’FOO). This returns three values: (1) The external form of the format spec for FOO (described in the next section), (2) the internal form of the format spec for FOO (described elsewhere), and (3) one of the symbols :EXTERNAL (meaning FOO had both an external and an internal format), :INTERNAL (meaning FOO had an internal format but no external one), or NIL (meaning FOO has no associated format). To associate a format with ’FOO, you say (SETF (GET-FORMAT ’FOO) FSPEC), where FSPEC is an external format spec (as described below). If you want to give FOO the same format as BAR, you can (of course) say (SETF (GET-FORMAT ’FOO) (GET-FORMAT ’BAR)), as in (SETF (GET-FORMAT ’DO*) (GET-FORMAT ’DO).

SEdit’s initial formatting information is broken into two parts: those formats which only exist in internal form and those which exist in both internal and external form. The internal-only formats are not of concern to us here (they really exist only for special interlisp constructs like CLISP). The external formats are all gotten by looking at entries on the *LISP-FORMAT-ALIST*, which we describe below. The function (RESET-FORMATS) resets SEdit’s formatting info to its initial state, and RESET-FORMATS is called as part of the initialization sequence. The function INSTALL-FORMAT-ALIST takes an alist with entries like those on the *LISP-FORMAT-ALIST* and installs the specified formatting information. INSTALL-FORMAT-ALIST also arranges that any calls to RESET-FORMATS will reinstall that formatting information in addition to the default information. It would be best, however, to get rid of format alists entirely and use defdefiners.

Documentation of Entries in the SEDIT:*LISP-FORMAT-ALIST*

Each entry in this list should be a symbol (or list of symbols) dotted with a format specification. The meaning of each entry (NAME . FSPEC) is: any list L whose car is NAME (or a member of NAME if it’s a list) should be formatted according to FSPEC.

Format Specifications

A format specification consists of an indentation specification (described below) followed by a bunch of options in PLIST format. The allowed options are:

:INLINE -- value can be T or NIL (default NIL). If T, the form will go all on one line if it fits. If NIL, the form will be broken across lines at arg boundaries even if it would all fit on one line. For example, OR has :INLINE T and LET has :INLINE NIL.

:MISER -- value can be :ALWAYS, :NEVER, or :TOFIT (default :TOFIT). Specifies when to use miser indentation. The default means use miser indentation if non-miser indentation would force the arguments into miser indentation.

:ARGS -- value should be a list of format specifications. These formats are assigned to the elements of the list L in order starting with the first element (which will be NAME). Note that these formats override any formats that would normally be assigned to the elements of L (based on their first elements). NIL is allowed in the :ARGS list, and means do *not* override the format of this element; that is, allow it to be formatted normally. Also, a symbol S is allowed in the :ARGS list if S has earlier been assigned a format; this means to assign S's format to this element. There are also two special keywords allowed as entries in the :ARGS list: :KEYWORD and :RECURSIVE. :KEYWORD means that if the element assigned this format is a symbol then treat it like a keyword, i.e., put it in bold face. (This list uses the convention that all symbols which allow declarations in their body [such as DO and LET] are formatted as keywords.) :RECURSIVE means to assign this element format FSPEC; that is, the entire top level format is assigned recursively to this element. This is very useful for formats like :DATA format. If L has more elements than there are entries in the :ARGS list, the last entry in the :ARGS list is repeated for all the extra elements of L. Hint: most :ARGS entries have NIL as their last element. If no :ARGS list is specified, the elements of L get their natural formatting.

:LAST -- value should be a format specification like those in the :ARGS list. This format specification will be applied to the last element of L but *only* if doing so would supercede the last entry in the :ARGS list. In other words, if the last element of L would receive the repeated format from the :ARGS list, it gets the :LAST format instead. This option is really only useful for pathologically formatted forms like Interlisp's SELECTQ.

:SUBLISTS -- value should be a list of element positions (counting from 1) or T. T means all of the arguments should be parsed as lists even if they are NIL (so NIL will display as () rather than NIL). A list of element position means those element positions will be parsed as lists. For example LET has :SUBLISTS (2) meaning the second element of a form whose first element is LET is a list (i.e., the binding list). DO has :SUBLISTS (2 3), DEFUN has :SUBLISTS (3) and COND has :SUBLISTS T. Default is :SUBLISTS NIL meaning print all NIL args as NIL not ().

Indentation Specifications

An indentation specification is either a symbol (normally a keyword) or a list. If it's a symbol, it's looked up on the SEDIT::*INDENT-ALIST* (which see) and the SEdit-internal indent specification found there is used. If it's a list, it consists of some optional keywords (described below) followed by argument group specifications. Each argument group specification is either a number or a list containing a single number. In both formats, the number indicates that that many arguments should be grouped together at a single indentation level. The simple number format means that each of those arguments should go on its own line (they will line up vertically with each other), while the number-in-a-list format means that the arguments in the group can go together on a single line if they fit. The indentation level for each argument group is determined by how many groups follow it in the indentation list. Each group is indented 1 level further in than the group which follows it; thus, the first argument group is indented most, the next one next most, and so on until the last one, which is always indented one step in from lambda-body level.

This is best explained with examples. A simple example is LET, whose indentation specification is (1). This means that LET will be followed by a single distinguished argument group consisting of one element (the binding list) which will be indented one step in from the let body. Another simple example is DO, whose indentation specification is (2). This means that DO will be followed by a single distinguished argument group consisting of two elements (the binding list and the termination clause) which will be indented one step in from the do body. It also means that the bindings and the termination will be required to go on separate lines. Contrast DO with DEFUN, whose indentation specification is ((2)). Like DO's spec, DEFUN's spec says there is one group with two members (the name and the lambda-list), but unlike DO's spec, DEFUN's spec says that the first two args can go on the same line if they fit there. Finally, consider a possible spec for MULTIPLE-VALUE-BIND of (1 1) which says that the first group consists of one arg (the variable list) and the second group consists of one arg (the form to eval). The form to eval will be indented one step in from the body, and the list of variables will be indented one step in from there.

Note that a group specification of 0 (zero) is allowed: this occupies an indentation step but does not put any arguments at that level. But we do not allow (0) as a group specification since this would not be any different than plain 0 and probably means that the specification is confused in some way.

The keywords allowed at the beginning of an indent specification are:

:BREAK or **:NOBREAK** or **:FIT --** These specify placement of the first argument in the first group. Default is :FIT, which means put this arg on the same line as the CAR of the form if it fits there in preferred mode, otherwise put it on the next line. Note that if the first arg goes on the same line as the CAR, its placement specifies the indentation level for the entire first group. That way long CARs will move the first group over to the right. (This makes the binding and termination of both DO and DO* line up, for example.) Specifying :NOBREAK means the first arg is forced to go on the same line as the CAR. Specifying :BREAK means the first arg is forced to go on the next line (and thus at the indentation level derived from the number of groups). UNWIND-PROTECT is a good example of using :BREAK to force the first arg onto its own line. Note that you can only specify one of :NOBREAK, :BREAK, or :FIT.

:TAGBODY -- Normally all forms in the body (whether atomic or not) go at the same indent level. Specifying :TAGBODY indicates that atomic body elements (*not* atomic elements of the argument groups) should be exdented to line up with the CAR of the entire list (such as PROG or TAGBODY, which see for examples).

:STEP -- This can be specified as many times as desired and each time increases the indentation of the body (and thus all the argument groups) by one step. If you just want to move some of the groups in but not all of them (and not the body) then use 0 groups at the appropriate place instead of using :STEP at the beginning. :STEP is very useful with :TAGBODY.

By the way, the normal body indentation is taken from the INDENT-BASE field of the LISP edit environment, which is initialized to the width of a capital ‘M’ in the SEdit default font. The normal indentation step is taken from the INDENT-STEP field of the LISP edit environment, which is initialized to twice the width of a capital ‘M’ (that is, twice INDENT-BASE). These defaults are chosen so that, in a fixed-width font, the body of a form lines up two characters in from the ‘(’ of the form, and each argument group line up two characters in from the next one (or the body). If you want non-standard values for either of these parameters, you can change the values in the LISP edit environment and then reinitialize your SEdit formats. Also, if you change font profiles, reinitializing SEdit will fix up the indents appropriately.

How Relinearization Works

The process by which SEdit optimizes formatting recomputation is strange and wonderful, so this is a long overdue attempt at explaining it.

We will start with a quick recap of SEdit's formatting model and the responsibilities of three node type methods: assign-format, compute-format-values, and linearize. We then describe the assumptions SEdit makes about when these have to be redone, and then describe the algorithm it uses to achieve this. We'll only go as far as getting the linear form fixed up; the step from there to updating the bits in the window is a whole 'nother story...

The formatting model

Linearization is the generation of a sequence of format tokens (space, string, bitmap, line break) from the internal tree representation of the data structure being edited. Doing a reasonable job for complex hierarchical structures involves a large number of constraints; SEdit uses a three pass algorithm to get its results:

formats: first, the presentation of a data structure often depends on the context in which it appears. for instance, a list occurring as the second element of a list beginning with let gets special treatment. another example is collapsing lists at a nesting level cut-off (actually, now that i think about it this would be much better done at parse time). SEdit currently does the first, but not the second (it ought to do both). to achieve this, each node can pass a 'format' to each of its subnodes. exactly what constitutes a format is arbitrary; it's up to the parent and child nodes to agree on what information will be passed (all current methods ignore format information they don't understand). at present lists pass their sublists list-format structures describing the appropriate presentation, and sometimes pass the atom :keyword to atomic elements which are to be printed in boldface.

each node type provides an assign-format method, which is called when the format of a node changes, and is repsonsible for propagating that change by calling assign-format on each of its subnodes; hence width estimates propagate from the top down.

width estimates: second, the presentation of a composite data structure will often depend on the size of its components. once the format information has been propagated to a node it will be asked to provide estimates of the size of its presentation, so that the nodes above it in the tree can plan their presentation intelligently. (note: unfortunately, most of the code calls width estimates "format values"; hence the method responsible is called compute-format-values). each node compute two numbers: inline-width and preferred-width. the inline width is the estimated width of this node's presentation assuming that there is room to fit it all on one line, or nil if the node's presentation will always span multiple lines. the preferred width is the width of the node's presentation assuming it will break at convenient spots. in computing these estimates a node needs access to the width estimates of its children; hence width estimates propagate from the bottom up.

linear form: finally, each node is asked to compute its linear form, by outputing a sequence of format tokens interspersed with the linear forms of its subnodes. the linearize method is told the horizontal position at which it is to begin, and the horizontal position of the right margin (which ought to try to stay within, but it's free to ignore). to get the best formatting linearization procedures generally have two formats: a preferred, reasonably-indented format and a tighter "miser" format, and choose the miser format whenever the width estimates indicate that the preferred format won't fit. each node makes this choice independently. the linear form is computed top down.

Incremental changes

The three-pass computation described above places (relatively) simple requirements on the methods, and suffices for a one-shot presentation. However, this is insufficient for SEdit; the presentation changes after each character typed, and repeating the above computation each time over the whole tree would clearly be unacceptably expensive. Instead, SEdit tries to determine the regions of the tree whose presentation might have changed given the editing operations performed, and calls the presentation methods only when the results might be different.

formats: in determining the format for its subnodes, a node is only allowed to base its decisions on its type, its own format, and the structure of it and its immediate subnodes. thus a list can change the format of its subnodes if it is edited, or one of its immediate subnodes is edited, but not if a nested subnode is edited. thus we only need to rerun a node's assign-format method if

- it was edited, or
- one of its subnodes was edited, or
- the assign-format method of its supernode was run (for one of these three reasons) and it assigned a different format to this node than previously

width estimates: the width estimates of a node must be determined based on its structure, its format, and the width estimates of its subnodes. thus we only need to rerun a node's compute-format-values method if

- it was edited, or
- its format changed, or
- the compute-format-values method of one of its subnodes was run (for one of these three reasons) and it changed that node's width estimates

linear form: the linear form of a node also depends on its structure, its format, and the width estimates of its subnodes. in addition, it can depend on the space between its starting horizontal position and the right margin. thus we only need to rerun a node's linearize method if

- it was edited, or
- its format changed, or
- the width estimates of any of its subnodes changed, or
- changes to one of its supernodes has caused its presentation to begin at a different horizontal position relative to the right margin

Relinearization

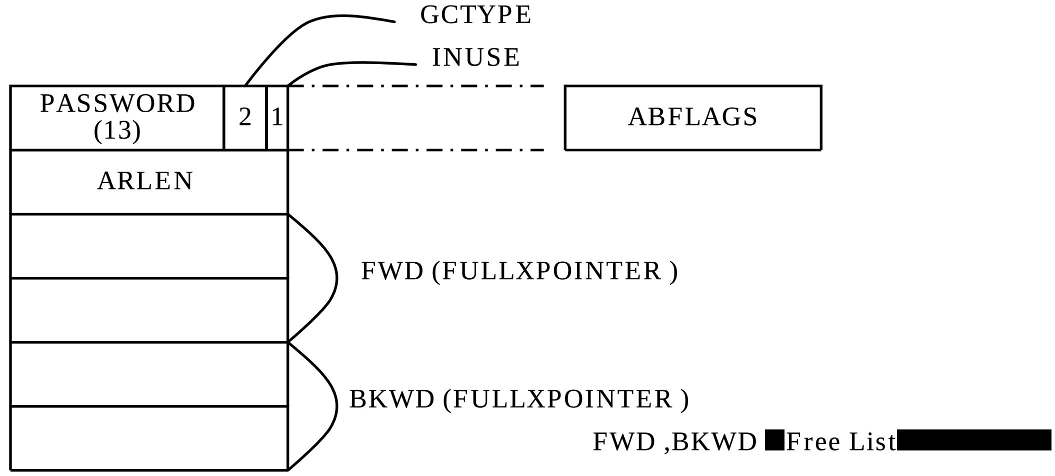
to implement the optimizations suggested above, SEdit must first know what parts of the tree have changed since it was last presented. therefore all procedures which change the tree structure are responsible for calling note-change on any node they change. also, all nodes added to the tree are marked as needing re-presentation. note-change inserts the changed node into a queue (the changed-nodes field in the edit-context) which is kept sorted by increasing depth. relinearize-where-necessary then implements the following algorithm:

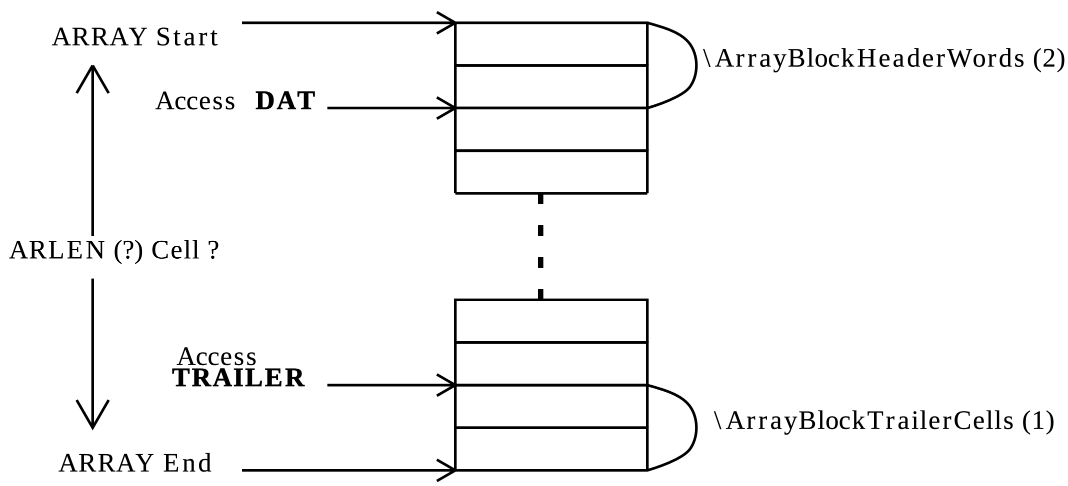
[inconsistency: note-change actually maintains the queue in order of decreasing depth (despite its comments to the contrary), and relinearize-where-necessary reverses it before it looks at it. this turns out to be fine, since it wants the queue in decreasing depth for the next step. ought to fix the comments though...]

1. for each node on the queue (from top to bottom), assign-format to its super node and add it to the queue (unless it's already in the queue) and then assign-format to it.
2. for each node now in the queue (from bottom to top) recompute its width estimates, and if they've changed push its super on the queue (where it will be picked up later in this step); if they don't change this is a point to start relinearizing from so push it onto another queue.
3. finally, reverse the new queue created in step 2 (so that it's now ordered from bottom to top) and for each node on it check to see that none of its super nodes are awaiting relinearization — if one is found, mark all the nodes between them changed so that the super's relinearization (yet to come) will include this node — otherwise just relinearize it.

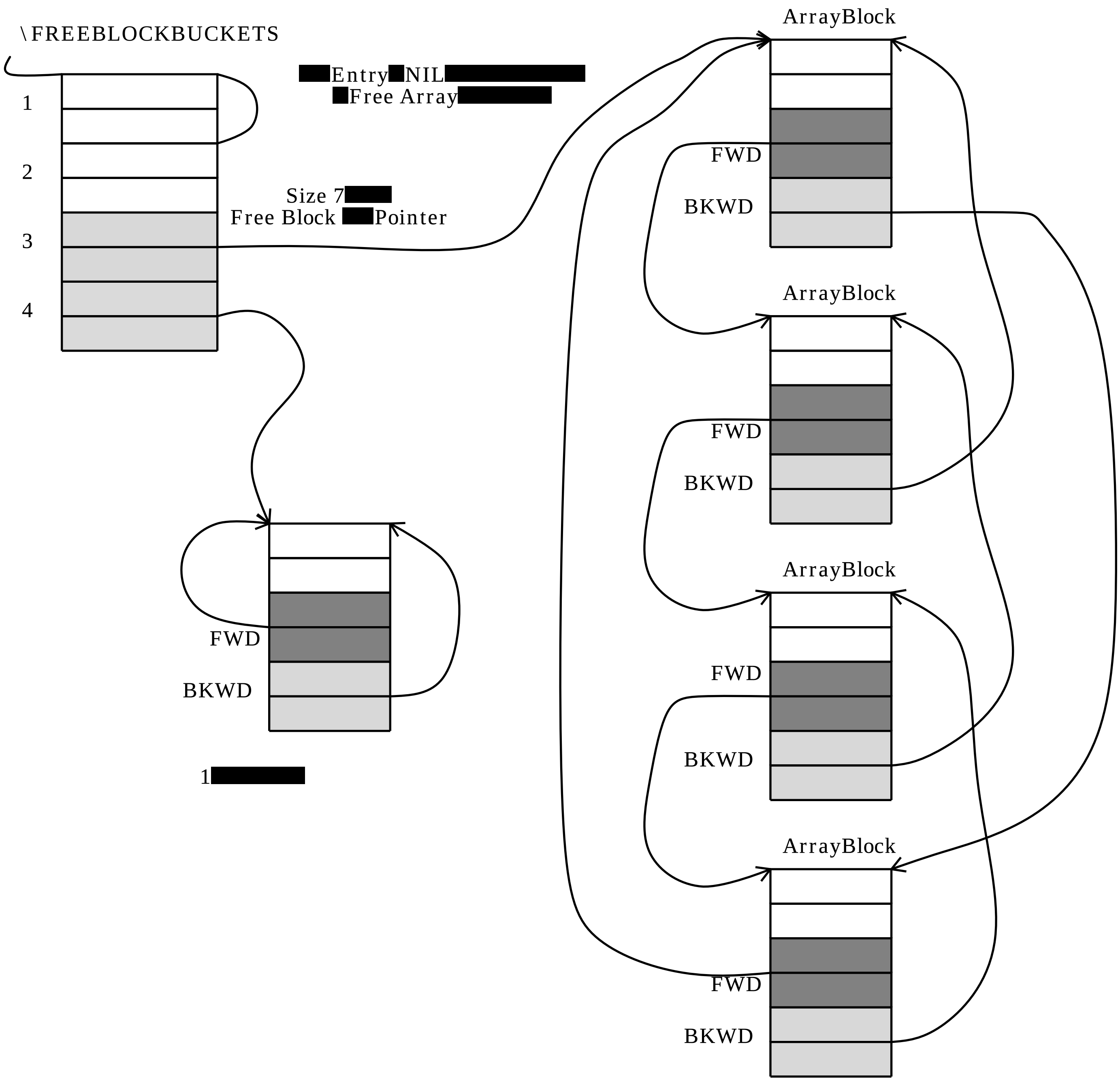
relinearizing a node is guaranteed to call its linearize method. In addition, it will call the linearize method of any subnode which (a) has changed, or (b) has been moved (relative to the right margin) so that it's linear form is no longer likely to be appropriate (the test for this is at the beginning of generate-linear-form) (and so on recursively into its subnodes). (and as i mentioned before, there's a whole extra story about how changes to the linear form are used to determine changes to the screen; this is yet to be documented at all). when relinearization of the original node is complete, an additional test is made: if the last line of the new linear form is a different length than the last line of the old linear form, this relinearization continues with the linear form of the super node, starting after the node just linearized (since the super may have made linearization decisions based on where that node ended). this process terminates when (a) a node's new linear form ends at the same horizontal position as before, or (b) the top of the tree is reached.

Structure of ARRAYBLOCK





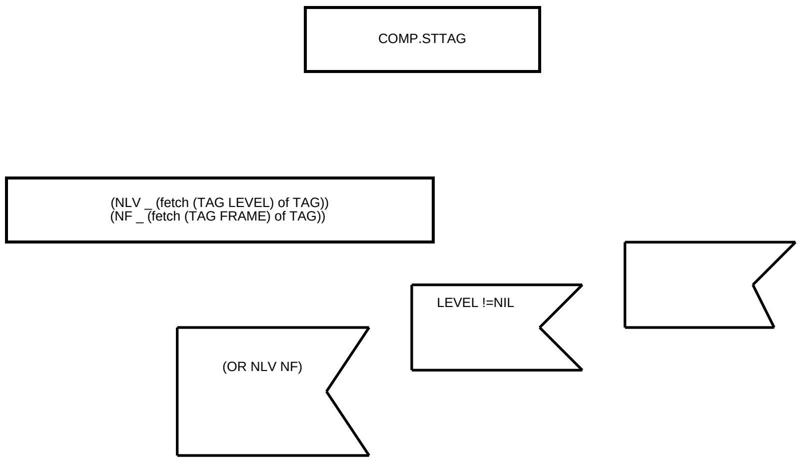
Array Free Block [redacted]



[redacted]

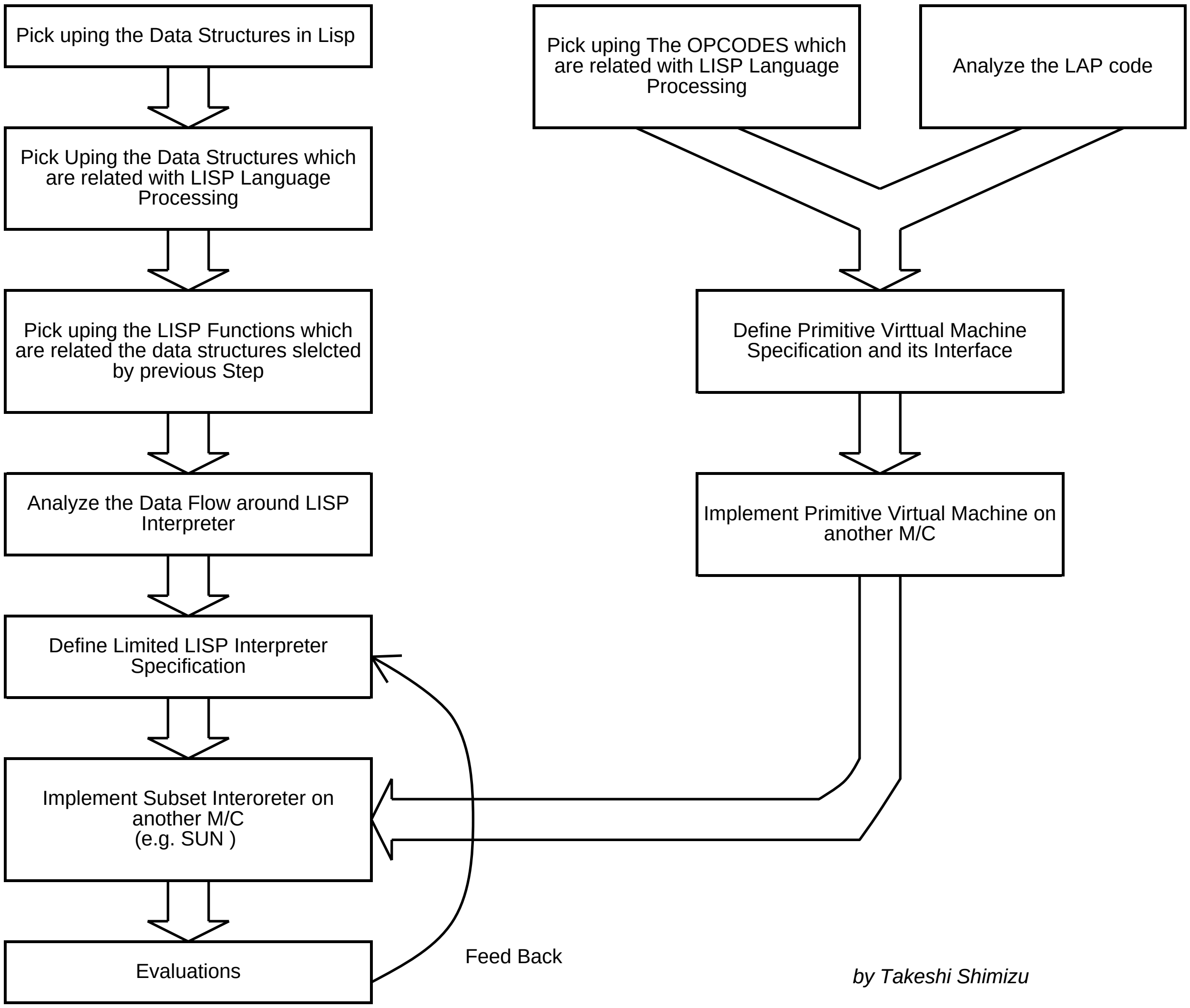
FWD[redacted]
\FREEBLOCKBUCKETS[redacted]
\FREEBLOCKBUCKETS[redacted]
Entry[redacted]FWD[redacted]

FWD[redacted]
\FREEBLOCKBUCKETS[redacted]MP.ERROR


COMP.STTAG
(NLV _ (fetch (TAG LEVEL) of TAG))
(NF _ (fetch (TAG FRAME) of TAG))
LEVEL !=NIL
(OR NLV NF)

No .		Func. Name		File Name	

ABSTRUCT		
	Type & Name	Comments
ARGS		
RETVAL		
LOCALS		
EXTERNS		



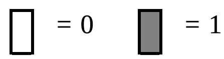
by Takeshi Shimizu

Interlisp-D Machine Development Concept (Primary Step)

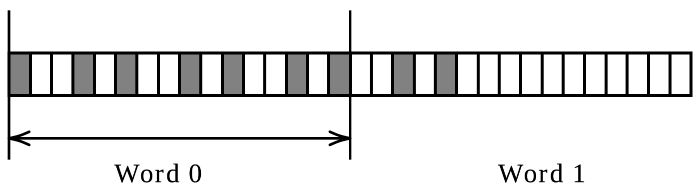
[redacted] [redacted]

Interlisp[redacted]BITMAP[redacted]

BitMap[redacted]

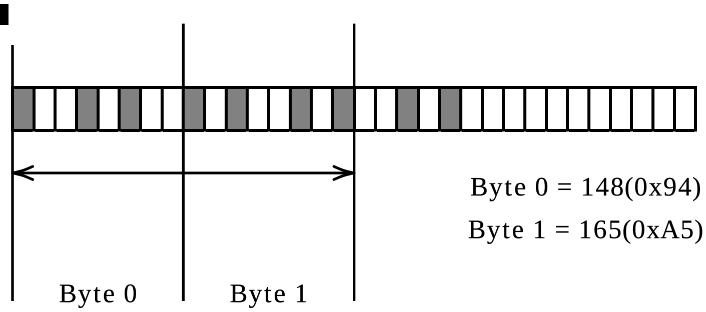


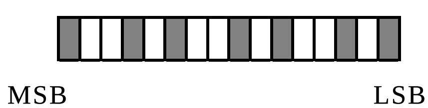
[redacted]

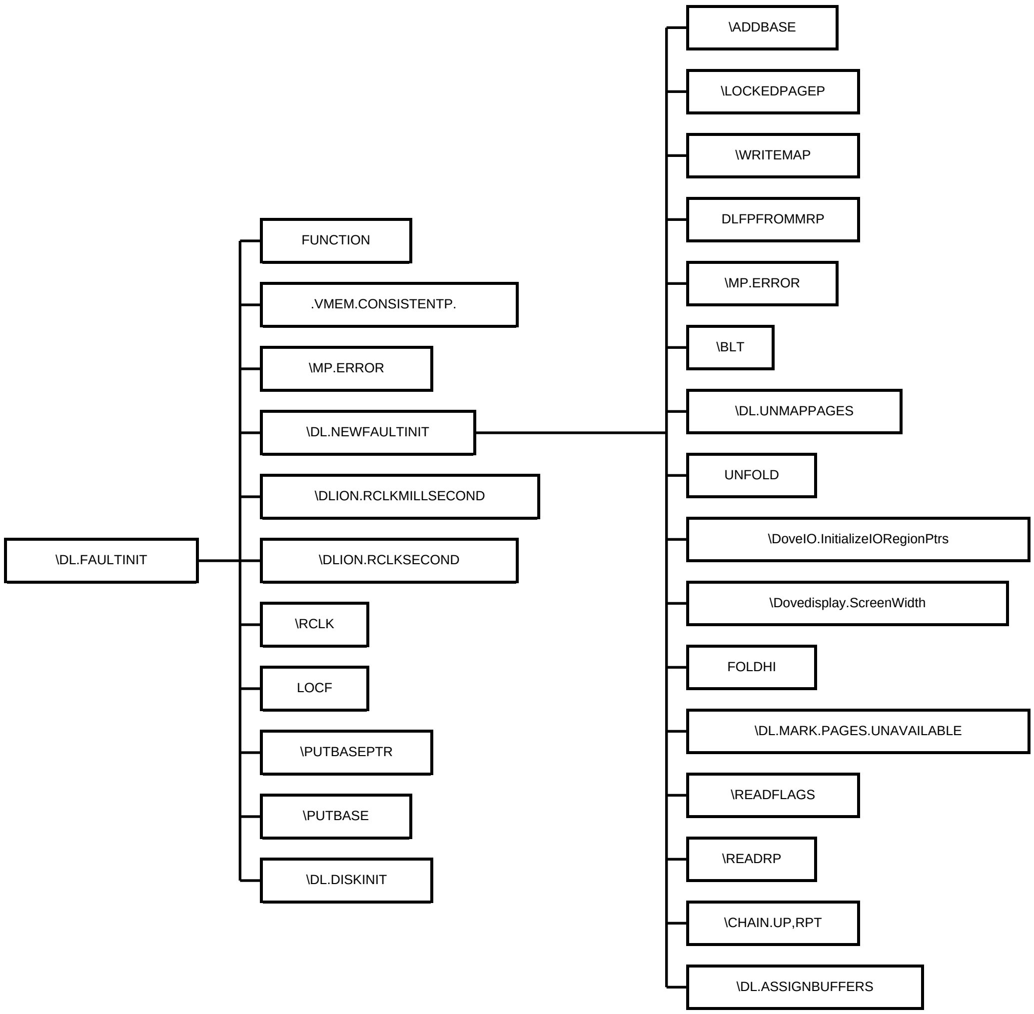


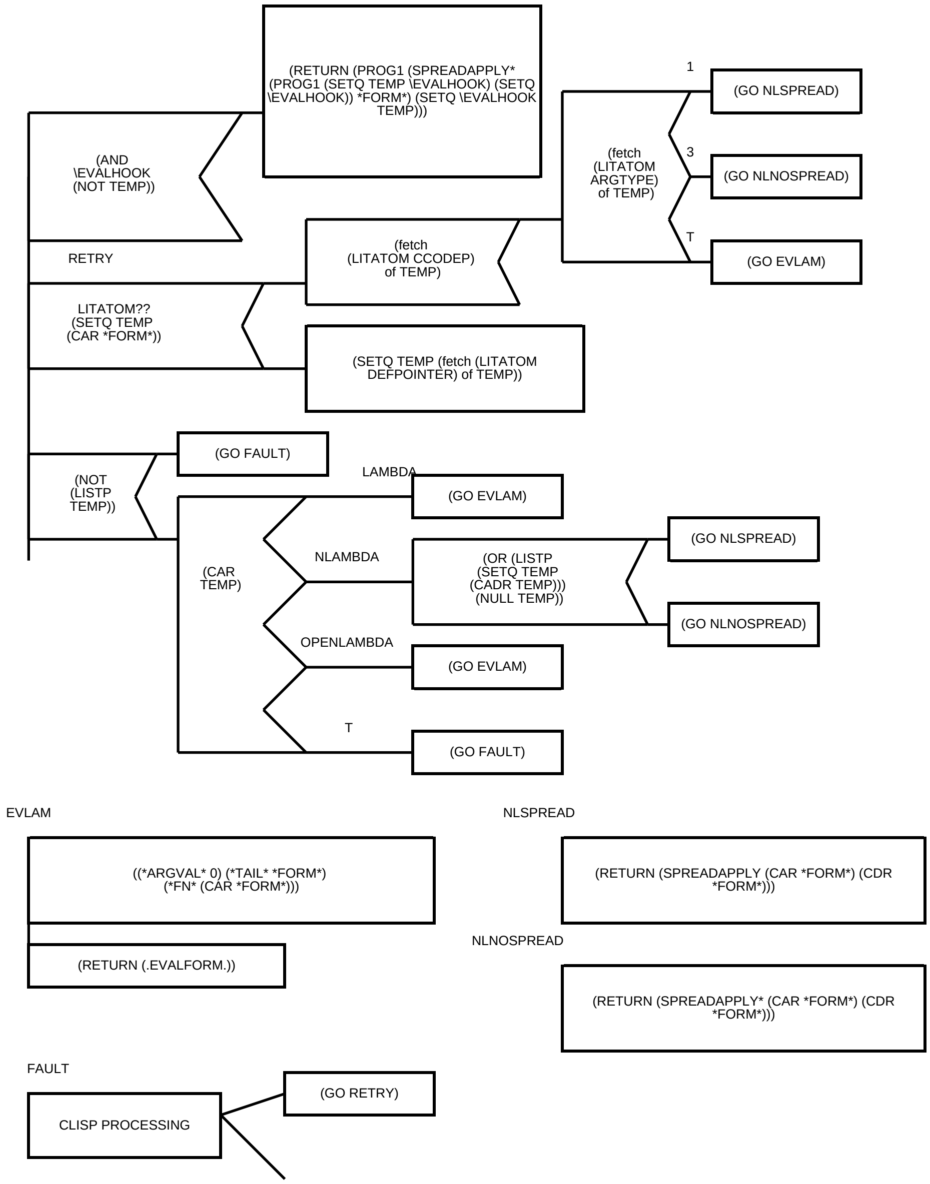
Word Access = 38053(0x94A5)

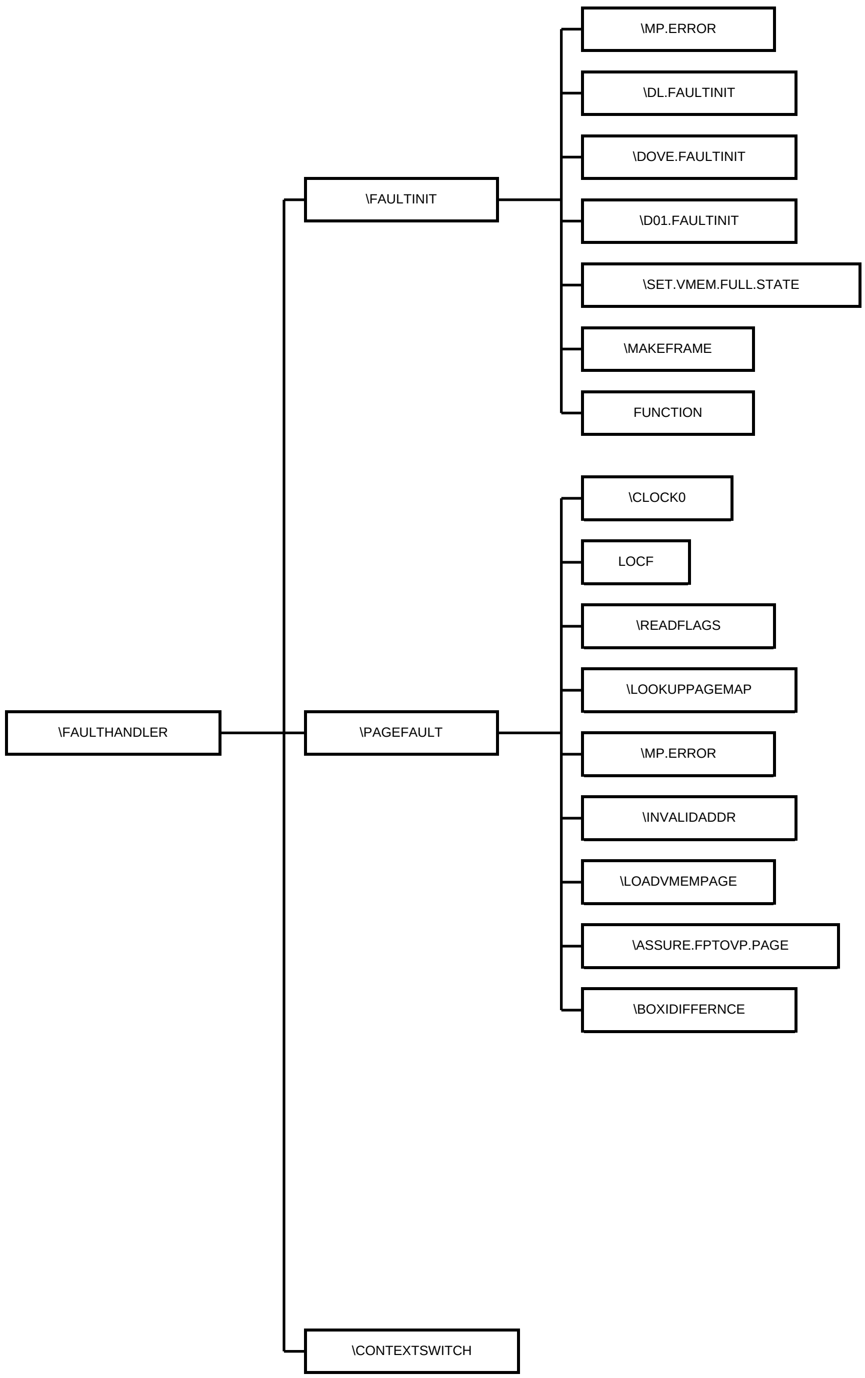
[redacted]

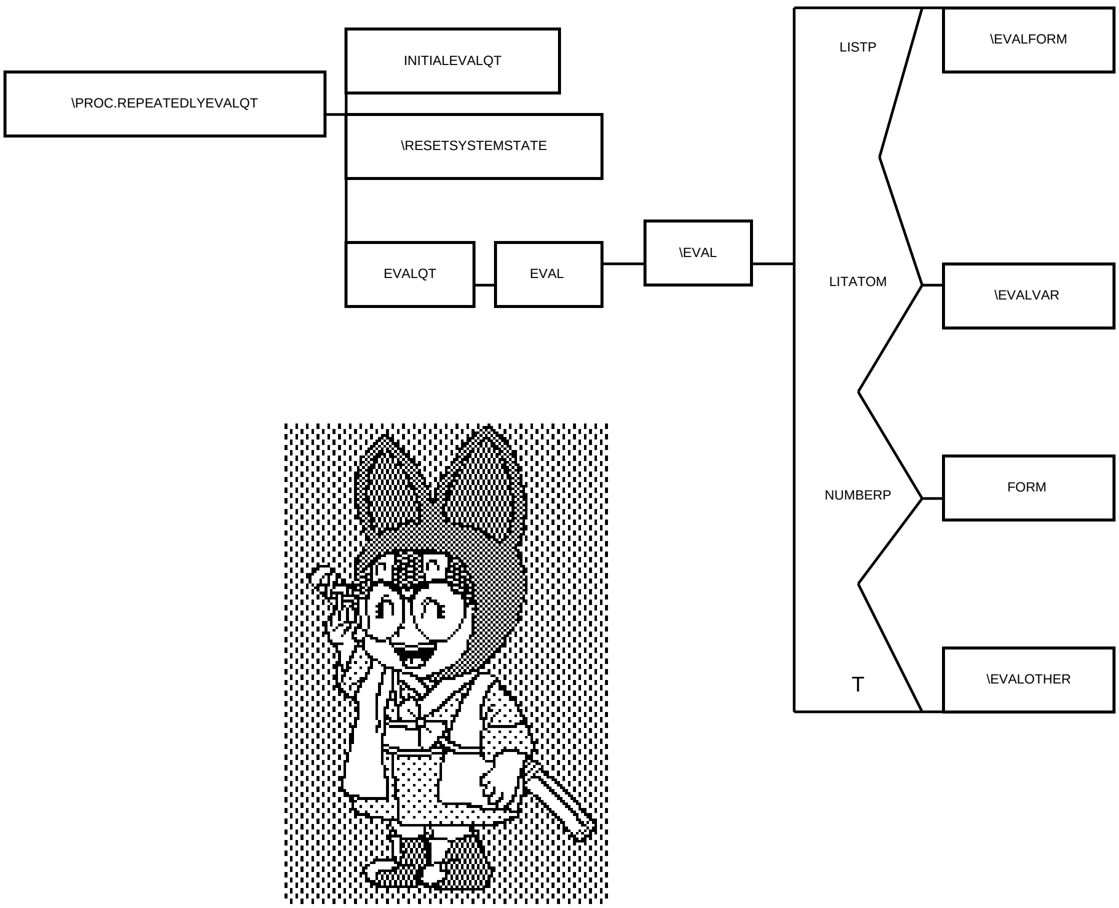





\DL.FAULTINIT
FUNCTION
.VMEM.CONSISTENTP.
\MP.ERROR
\DL.NEWFAULTINIT
\DLION.RCLKMILLSECOND
\DLION.RCLKSECOND
\RCLK
LOCF
\PUTBASEPTR
\PUTBASE
\DL.DISKINIT
\ADDBASE
\LOCKEDPAGEP
\WRITEMAP
DLFPFROMMRP
\MP.ERROR
\BLT
\DL.UNMAPPAGES
UNFOLD
\DoveIO.InitializeIORegionPtrs
\Dovedisplay.ScreenWidth
FOLDHI
\DL.MARK.PAGES.UNAVAILABLE
\READFLAGS
\READRP
\CHAIN.UP,RPT
\DL.ASSIGNBUFFERS

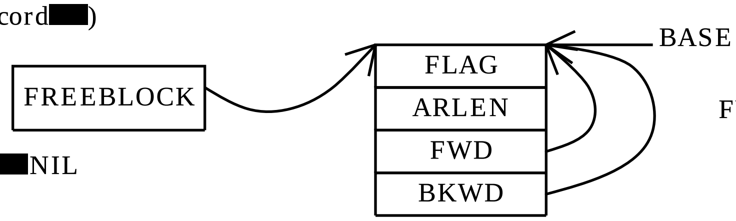

(RETURN (PROG1 (SPREADAPPLY* (PROG1 (SETQ TEMP \EVALHOOK) (SETQ \EVALHOOK)) *FORM*) (SETQ \EVALHOOK TEMP)))
(AND \EVALHOOK (NOT TEMP))
1
(GO NLSPREAD)
3
(GO NLNOSPREAD)
T
(GO EVLAM)
(fetch (LITATOM ARGTYPE) of TEMP)
RETRY
(fetch (LITATOM CCODEP) of TEMP)
LITATOM?? (SETQ TEMP (CAR *FORM*))
(SETQ TEMP (fetch (LITATOM DEFPOINTER) of TEMP))
(GO FAULT)
(NOT (LISTP TEMP))
LAMBDA
(GO EVLAM)
(CAR TEMP)
NLAMBDA
(OR (LISTP (SETQ TEMP (CADR TEMP))) (NULL TEMP))
(GO NLSPREAD)
(GO NLNOSPREAD)
OPENLAMBDA
(GO EVLAM)
T
(GO FAULT)
EVLAM
((*ARGVAL* 0) (*TAIL* *FORM*) (*FN* (CAR *FORM*)))
(RETURN (.EVALFORM.))
NLSPREAD
(RETURN (SPREADAPPLY (CAR *FORM*) (CDR *FORM*)))
NLNOSPREAD
(RETURN (SPREADAPPLY* (CAR *FORM*) (CDR *FORM*)))
FAULT
CLISP PROCESSING
(GO RETRY)


\FAULTHANDLER
\FAULTINIT
\MP.ERROR
\DL.FAULTINIT
\DOVE.FAULTINIT
\D01.FAULTINIT
\SET.VMEM.FULL.STATE
\MAKEFRAME
FUNCTION
\PAGEFAULT
\CLOCK0
LOCF
\READFLAGS
\LOOKUPPAGEMAP
\MP.ERROR
\INVALIDADDR
\LOADVMEMPAGE
\ASSURE.FPTOVP.PAGE
\BOXIDIFFERNCE
\CONTEXTSWITCH


\PROC.REPEATEDLYEVALQT
INITIALEVALQT
\RESETSYSTEMSTATE
EVALQT
EVAL
\EVAL
LISTP
LITATOM
NUMBERP
T
\EVALFORM
\EVALVAR
FORM
\EVALOTHER

\LINKBLOCK

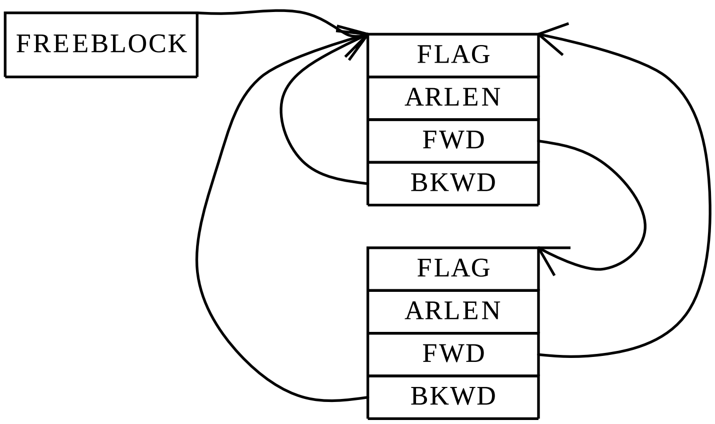
Case 1 ([redacted]Record[redacted])

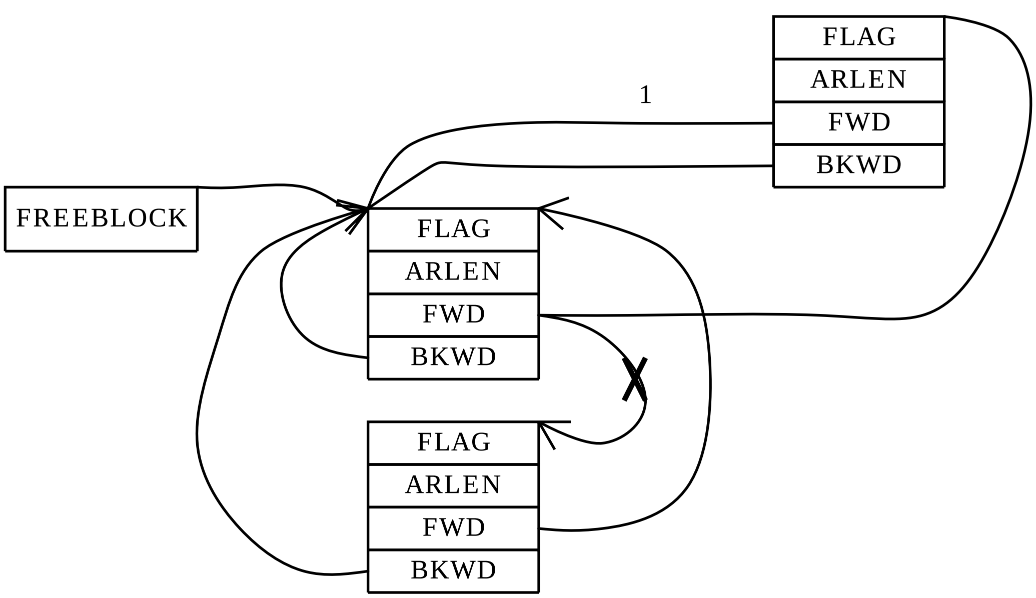


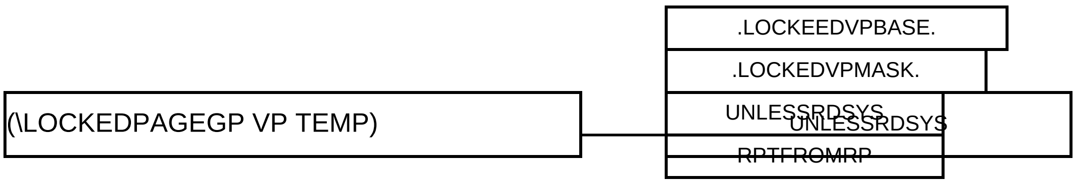
[redacted]NIL

FWD,BKWD[redacted]

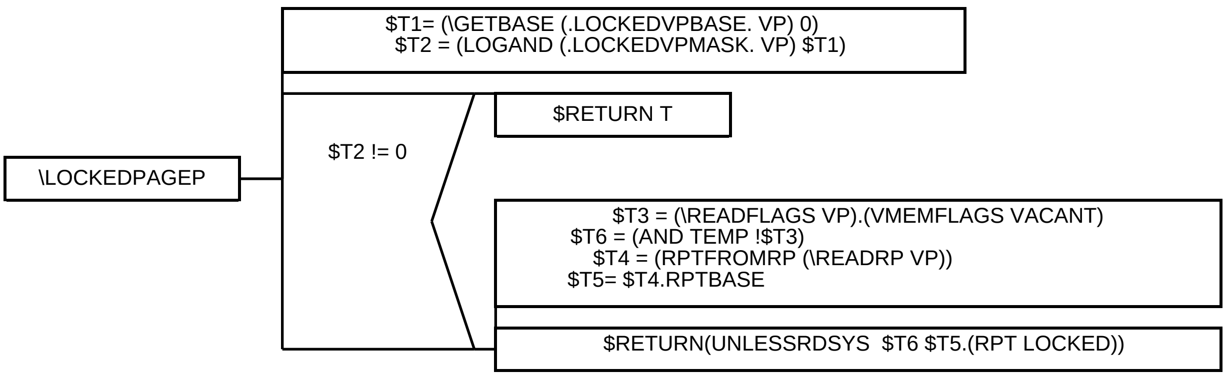
Case 2 LINK[redacted]







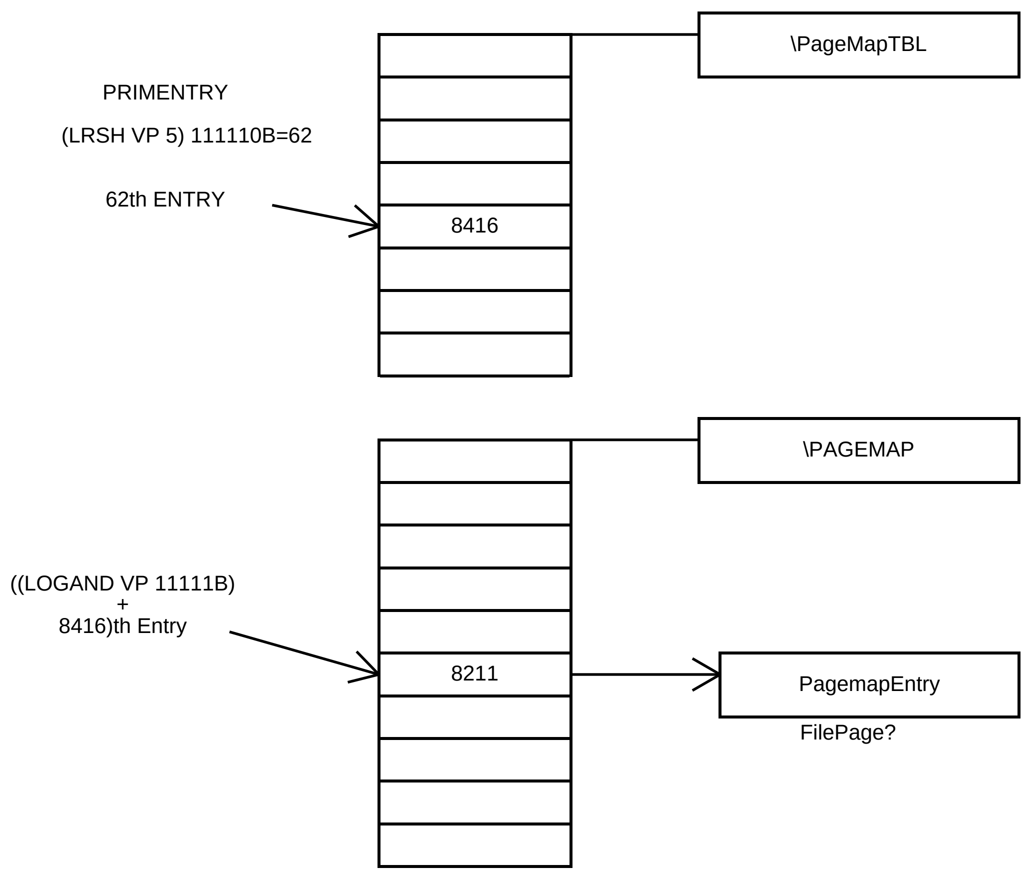
(\LOCKEDPAGEGP VP TEMP)
.LOCKEEDVPBASE.
.LOCKEDVPMASK.
UNLESSRDSYS
UNLESSRDSYS
RPTFROMRP



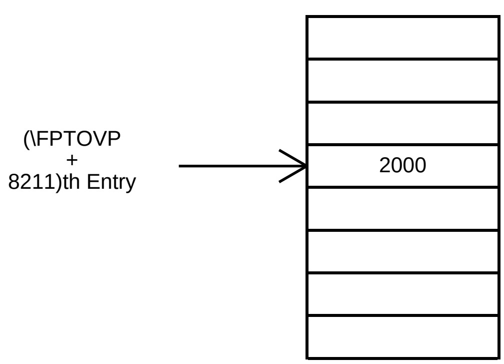
\LOCKEDPAGEP
$T1= (\GETBASE (.LOCKEDVPBASE. VP) 0)
$T2 = (LOGAND (.LOCKEDVPMASK. VP) $T1)
$T2 != 0
$RETURN T
$T3 = (\READFLAGS VP).(VMEMFLAGS VACANT)
$T6 = (AND TEMP !$T3)
$T4 = (RPTFROMRP (\READRP VP))
$T5= $T4.RPTBASE
$RETURN(UNLESSRDSYS $T6 $T5.(RPT LOCKED))

\LOOKUPPAGEMAP

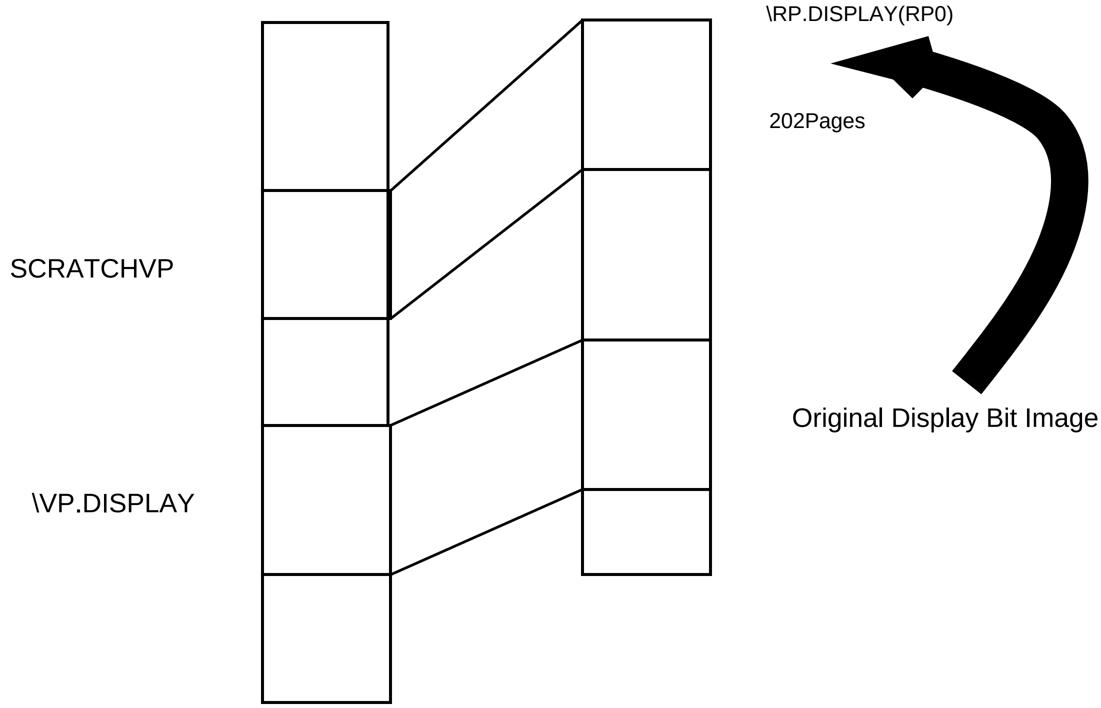
VP= 11111010000B=2000



Operations by Using \FPTOVP

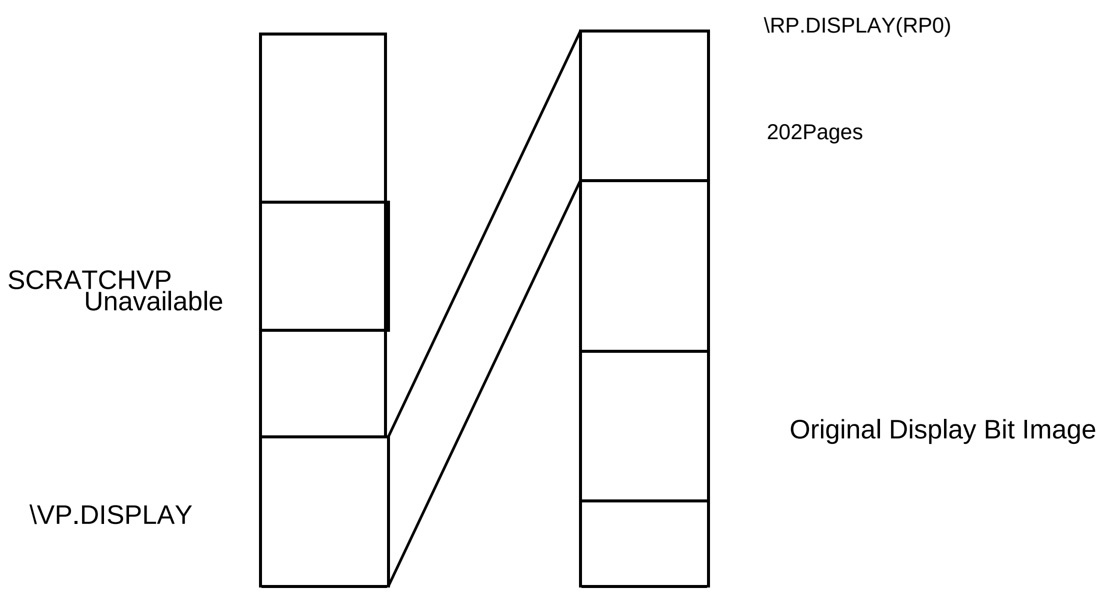


MOVE DIsplay Region First

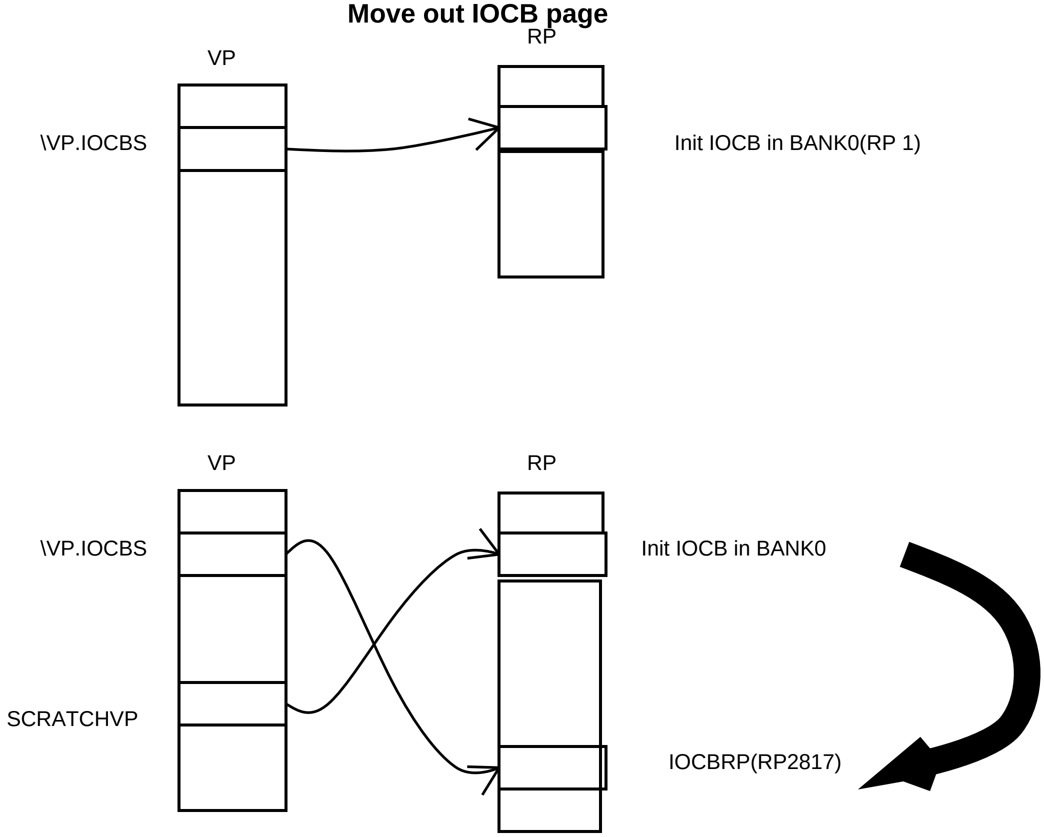


(\BLT SCRATCHBASE (create POINTER #PAGE _\VP.DISPLAY) NWORD)

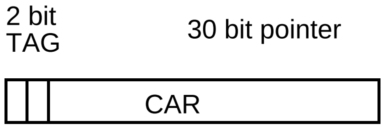
MOVE DIsplay Region Second



Move out IOCB page



(\BLT \IOCBPAGE SCRATCHBASE WORDSPERPAGE)

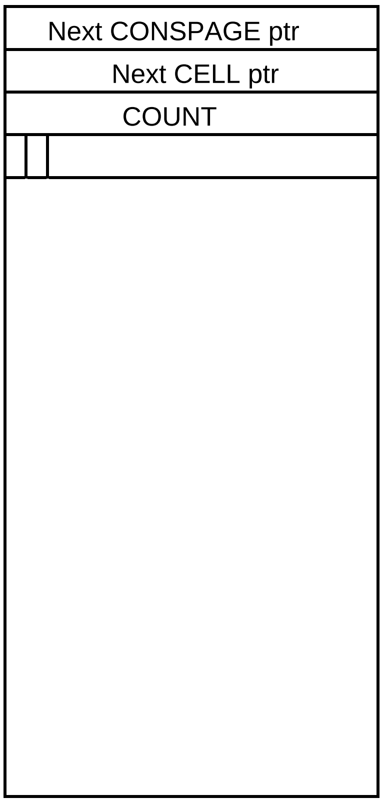


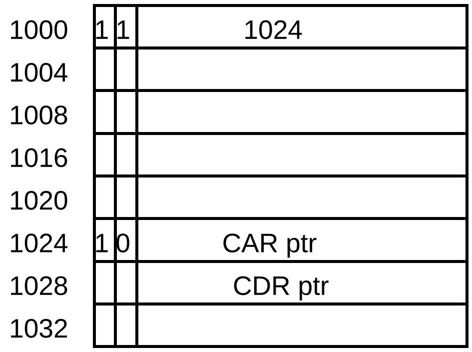
00 : CDR is NIL

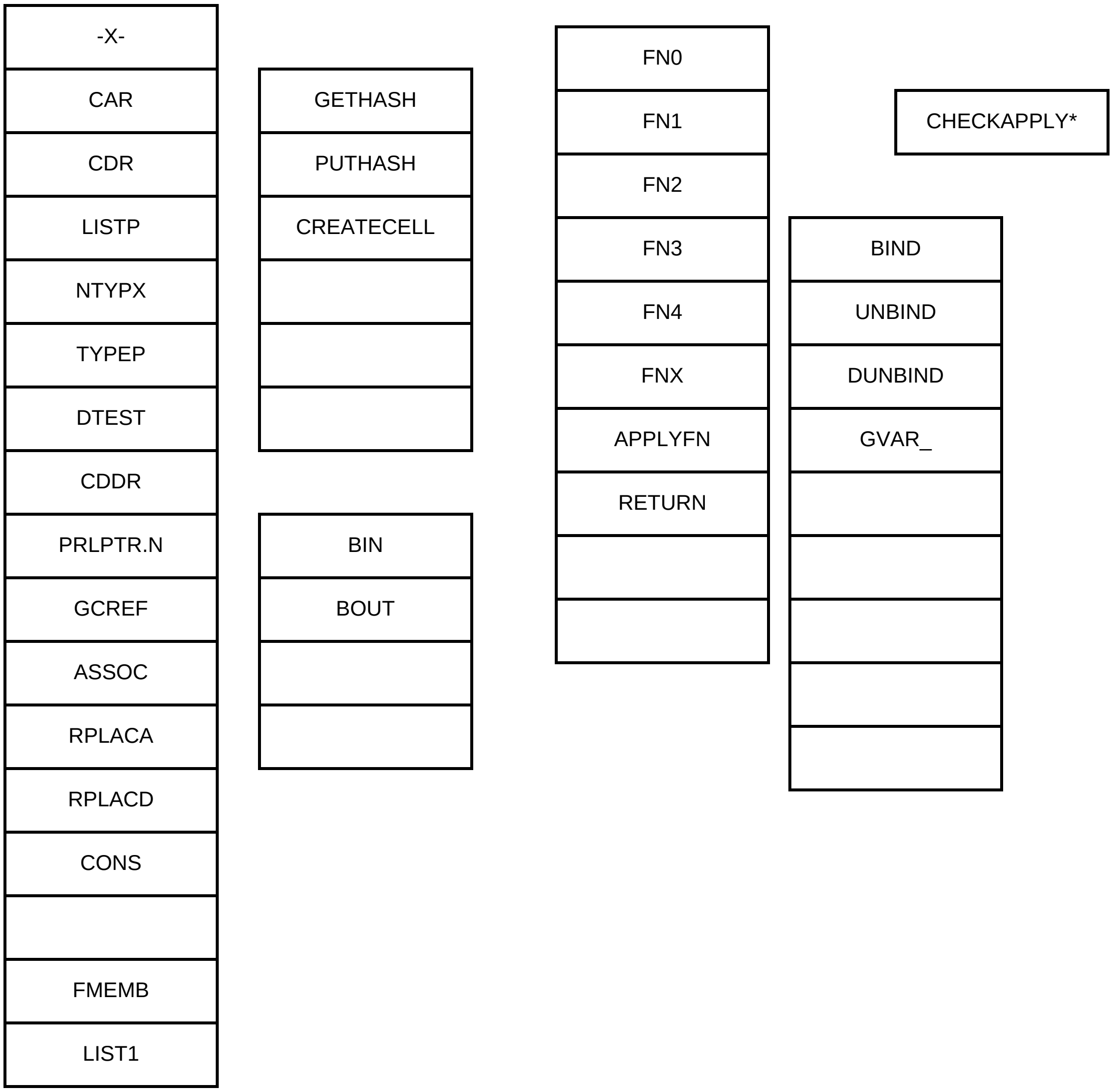
01 : CDR is PREV

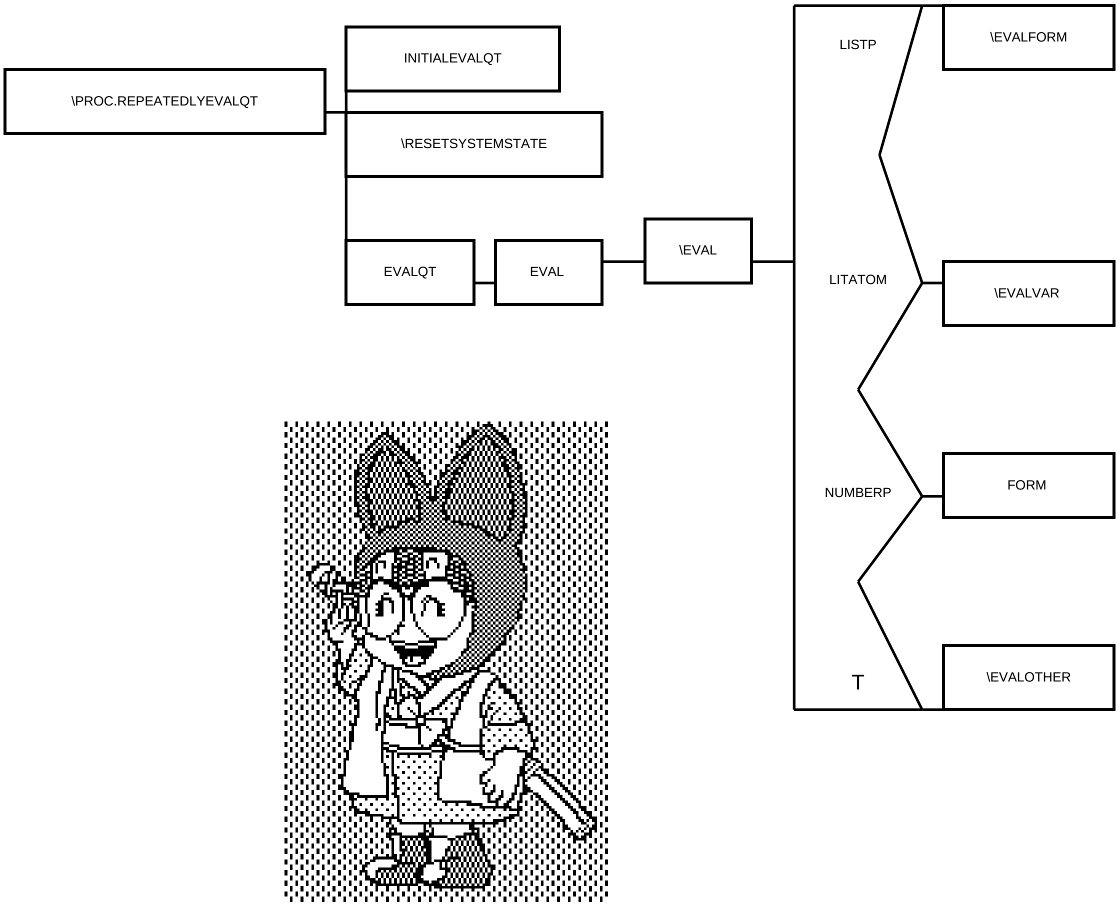
10 : CDR is NEXT

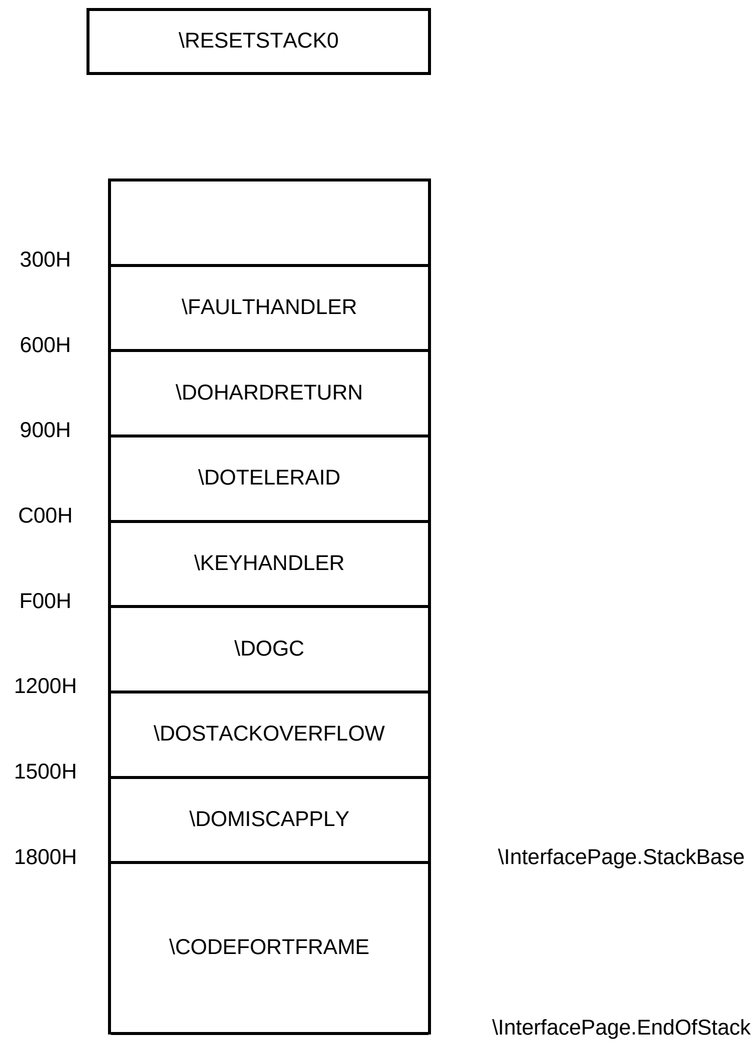
11 : CDR INDIRECT



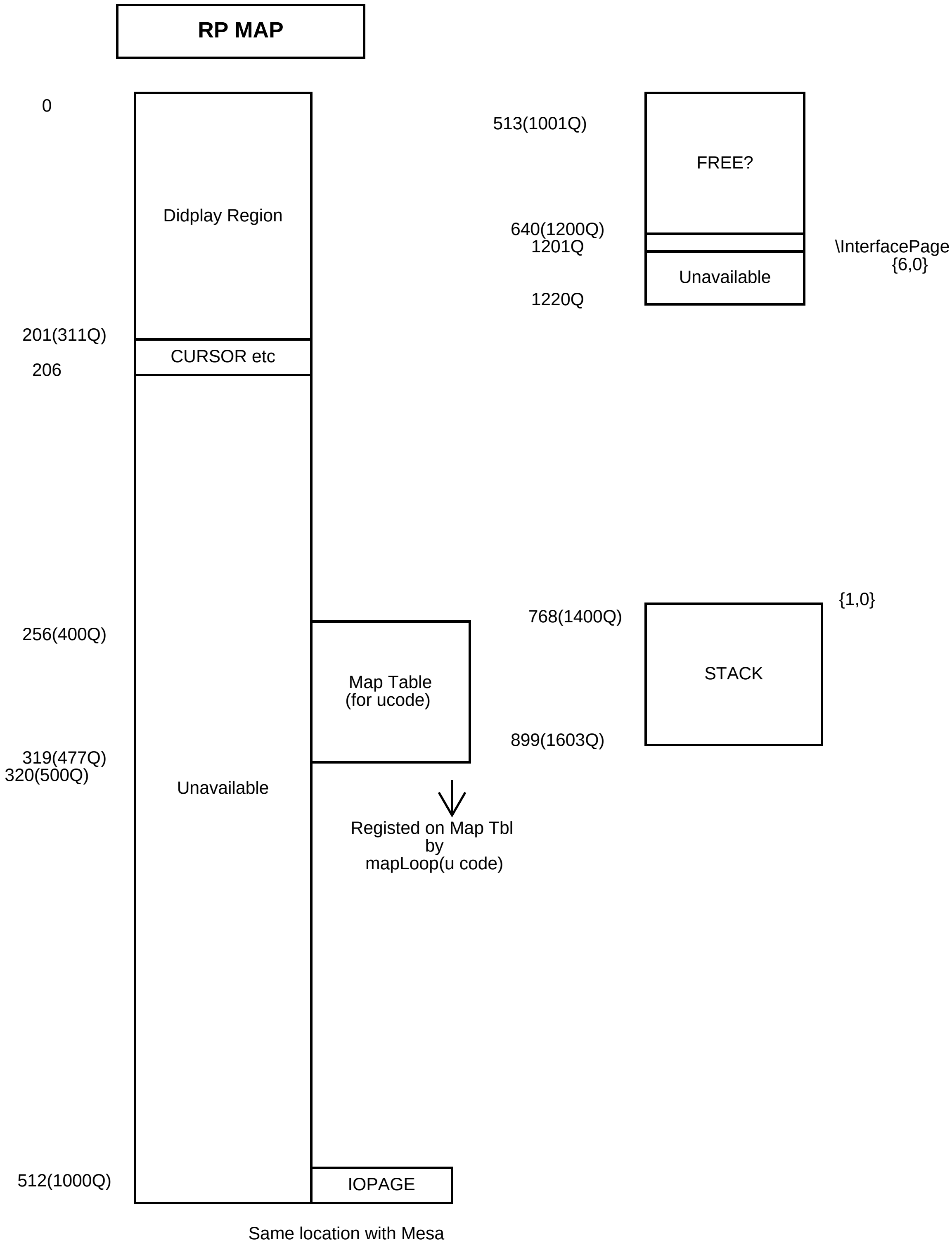



-X-
CAR
CDR
LISTP
NTYPX
TYPEP
DTEST
CDDR
PRLPTR.N
GCREF
ASSOC
RPLACA
RPLACD
CONS
FMEMB
LIST1
GETHASH
PUTHASH
CREATECELL
BIN
BOUT
FN0
FN1
FN2
FN3
FN4
FNX
APPLYFN
RETURN
BIND
UNBIND
DUNBIND
GVAR_
CHECKAPPLY*


\PROC.REPEATEDLYEVALQT
INITIALEVALQT
\RESETSYSTEMSTATE
EVALQT
EVAL
\EVAL
LISTP
LITATOM
NUMBERP
T
\EVALFORM
\EVALVAR
FORM
\EVALOTHER



\InterfacePage.FAULTFXP =302H
\InterfacePage.HardReturnFXP =602H
\InterfacePage.TELERAIDFXP =902H
\InterfacePage.KbdFXP =C02H
\InterfacePage.GCFXP =F02H
\InterfacePage.SubovFXP =1202H
\InterfacePage.MiscFXP =1502H
\InterfacePage.ResetFXP =1802H


RP MAP
0
Didplay Region
201(311Q)
CURSOR etc
206
256(400Q)
Map Table
(for ucode)
319(477Q)
320(500Q)
Unavailable
Registed on Map Tbl
by
mapLoop(u code)
512(1000Q)
IOPAGE
Same location with Mesa
513(1001Q)
FREE?
640(1200Q)
1201Q
Unavailable
1220Q
\InterfacePage
{6,0}
768(1400Q)
STACK
899(1603Q)
{1,0}

1024(2000Q)
FPTOVP
1061(2045Q) {4,45}

4096(10000Q)
Unavailable | \REALPAGETABLE
4352(10400Q)

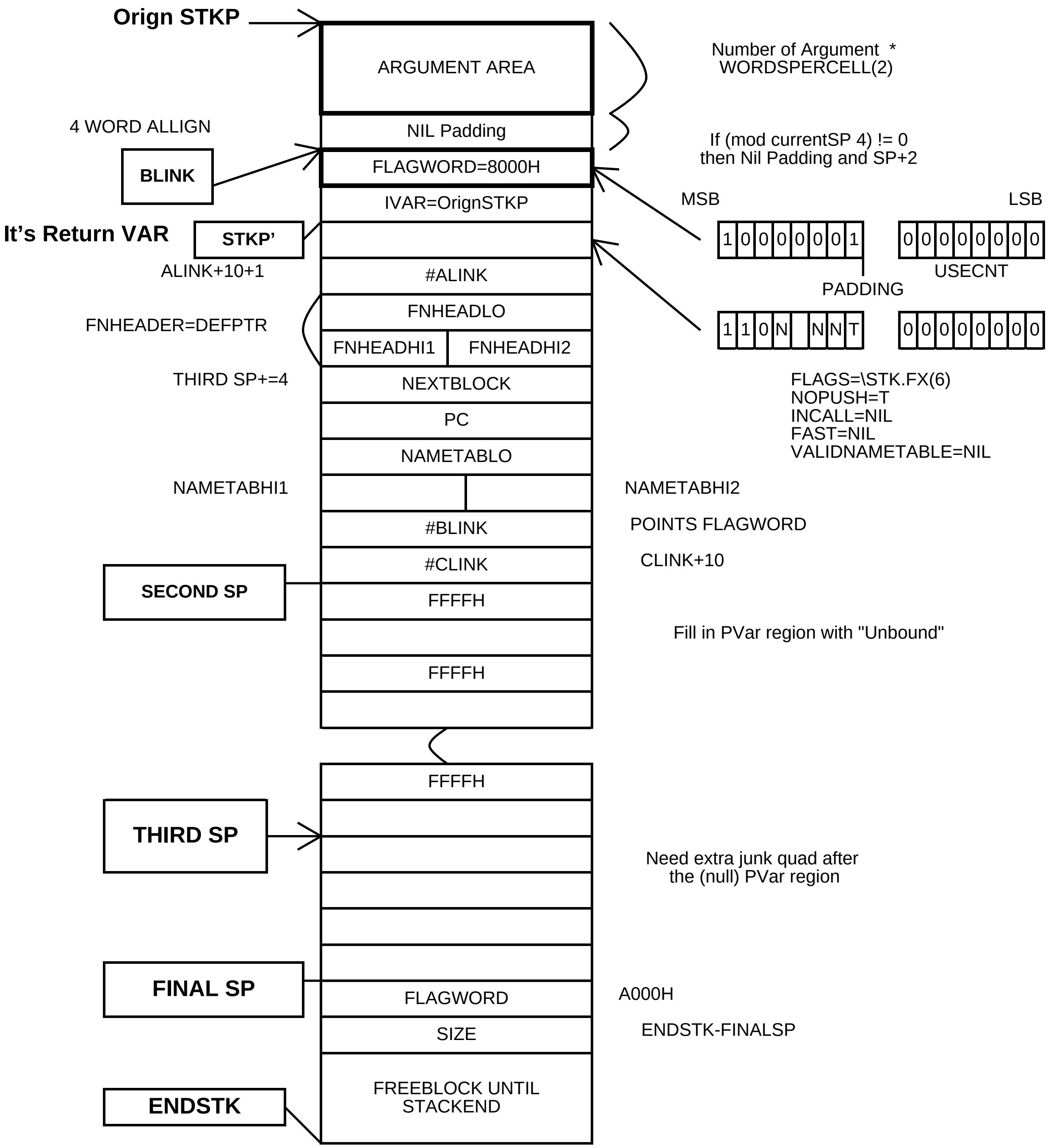
1289(2400Q) {6,200}
TYPETABLE
1408(2600Q) {6,377}
GCTABLE {20,0}
1536(3000Q)
GCOVERFLOW

1780(3364Q) {6,2}
\PagemapTBL
1788(3374Q) {6,11}
{5,0}
\PAGEMAP
1799(3407Q) {5,13}
{6,160}-{6,177}
1815(3427Q)
{6,12}-{6,15}
{6,20}
1820(3434Q)
1821(3435Q)
\DEFSPACE
1832(3450Q)
\VALSPACE
1840(3460Q) {21,0}\HTCOLL
1841(3461Q)
{23,56}-
{23,340}
1952(3640Q)
{24,75}-{24,123},{24,373}
{24,44}-
{24,50},
{24,53}-{24,56}

4225	2046 {4,46}	2045	Locked	Dirty
3346	2447 {5,47}	4060	Locked	Dirty
4421	2052 {4,52}	2051	Locked	Dirty
4501	2453 {5,53}	4064	Locked	Dirty
5450	2454 {5,54}	4065	Locked	Dirty
11553	2450 {5,50}	4061	Locked	Dirty
11603	2047 {4,47}	2046	Locked	Dirty
13716	2050 {4,50}	2047	Locked	Dirty
13762	2451 {5,51}	4062	Locked	Dirty
15531	2051 {4,51}	2050	Locked	Dirty
15627	2452 {5,52}	4063	Locked	Dirty

2081(4041Q)
{5,27}-{5,46}
2096(4060Q)

2817(5401Q)
Unavailable
\IOCBPAGE(Depends on MEM size)


Orign STKP
ARGUMENT AREA
Number of Argument *
WORDSPERCELL(2)
4 WORD ALLIGN
NIL Padding
If (mod currentSP 4) != 0
then Nil Padding and SP+2
BLINK
FLAGWORD=8000H
IVAR=OrignSTKP
MSB
LSB
It's Return VAR
STKP'
1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
ALINK+10+1
#ALINK
USECNT
PADDING
FNHEADLO
FNHEADER=DEFPTR
FNHEADHI1
FNHEADHI2
1 1 0 N N N T
0 0 0 0 0 0 0 0
THIRD SP+=4
NEXTBLOCK
FLAGS=\STK.FX(6)
NOPUSH=T
INCALL=NIL
FAST=NIL
VALIDNAMETABLE=NIL
PC
NAMETABLO
NAMETABHI1
NAMETABHI2
#BLINK
POINTS FLAGWORD
#CLINK
CLINK+10
SECOND SP
FFFFH
Fill in PVar region with "Unbound"
FFFFH
FFFFH
THIRD SP
Need extra junk quad after
the (null) PVar region
FINAL SP
FLAGWORD
A000H
SIZE
ENDSTK-FINALSP
ENDSTK
FREEBLOCK UNTIL
STACKEND

FUNC NAME	\LOOKUPPAGEMAP

ID	

CALLING SEQUANCE

(\LOOKUUPPAGEMAP VP)

LOCAL VAL

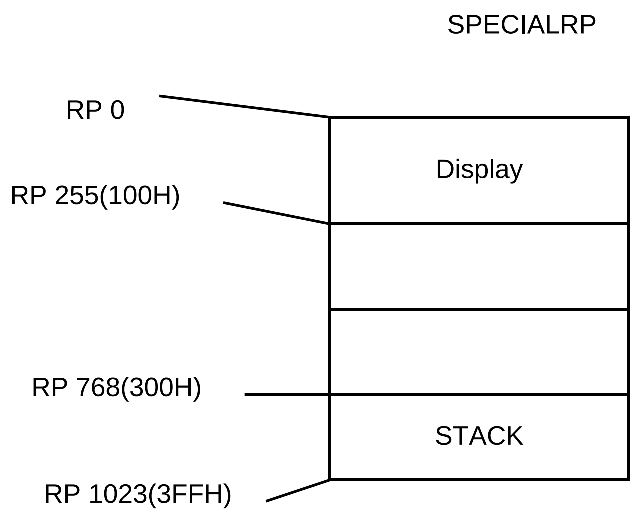
PRIMENTRY ** (\GETBASE \PageMapTBL (fetch (VP PRIMARYKEY) of VP]

GLOOVAL VAL

NAME	TYPE
\PageMapTBL (R) \EMPTYPMTENTRY (R) \PAGEMAP (R) VP (R)	RECORD VP

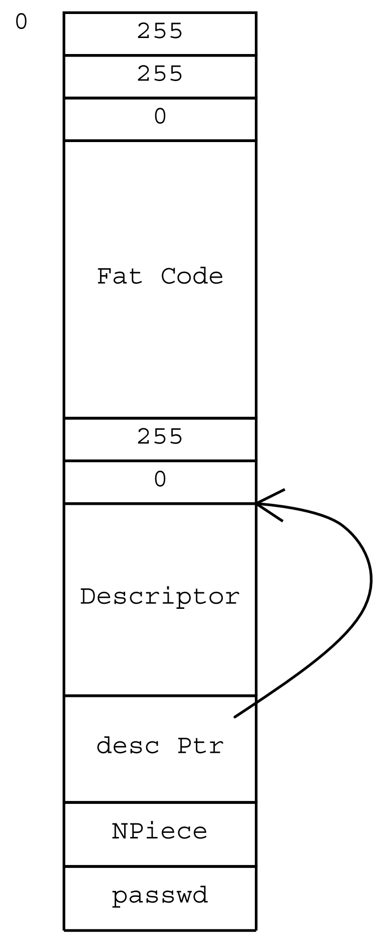
CALL FUNCTIONS

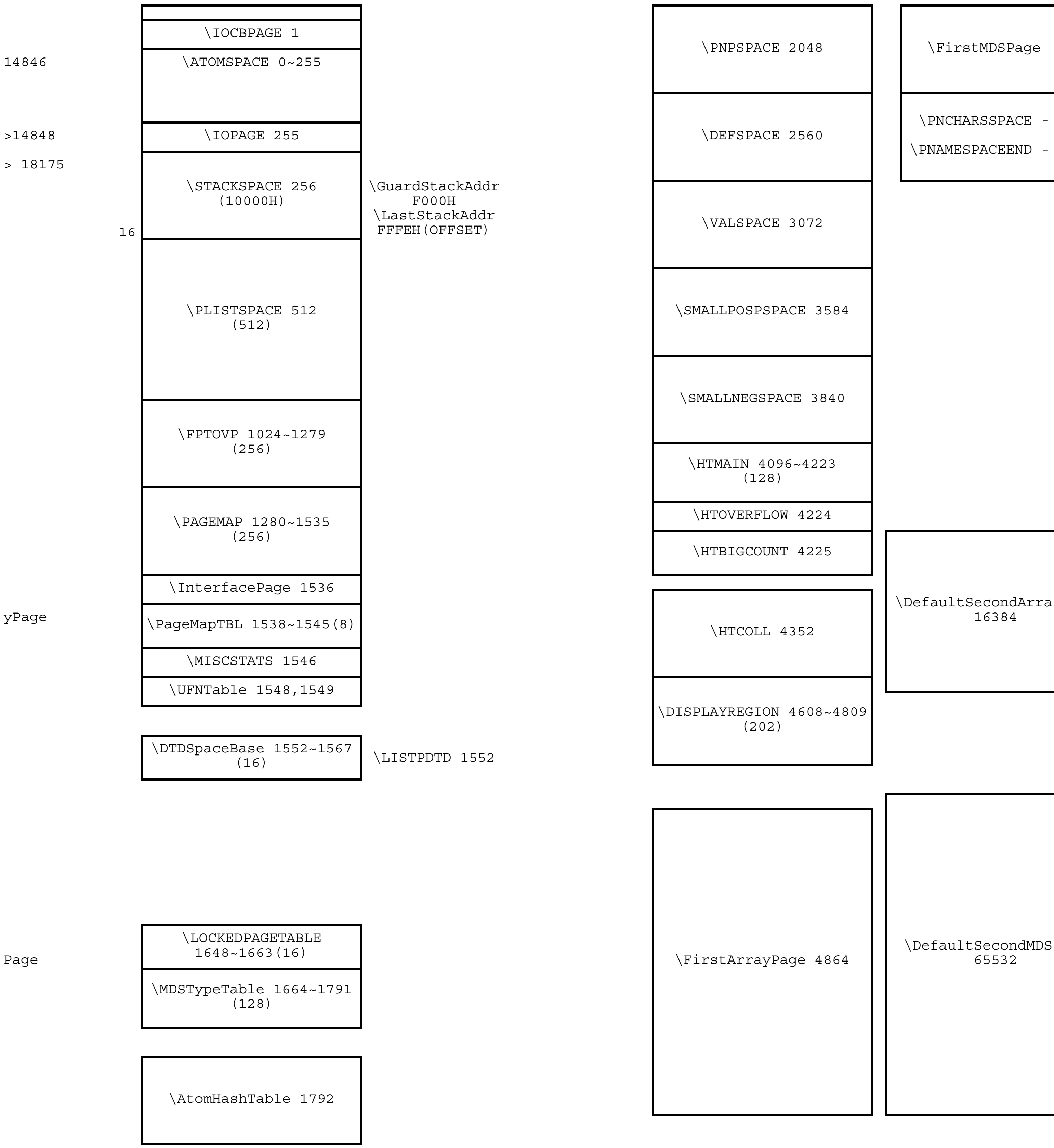
\GETBASE


SPECIALRP
RP 0
Display
RP 255(100H)
RP 768(300H)
STACK
RP 1023(3FFH)

Memory Stats on STARTUP

Address	Region
0,100H	\IOCBPAGE
0,300H -0,23FEH	?STACKAREA?
0,FF00H	\ IOPAGE
4,0000H	\FPTOVP
5,000H	\PAGEMAP 256 Pages
6,0000H	\InterfacePage
6,0200H	\PageMapTBL
6,0A00H	\MISCSTATS
6,0C00H	\UFNTable
7,0000H	AtomHashTable
AH,0000	\VMEMPAGEP
12H,0000	\DISPLAYREGION


0
255
255
0
Fat Code
255
0
Descriptor
desc Ptr
NPiece
passwd



14846
>14848
> 18175
16
yPage
Page
\IOCBPAGE 1
\ATOMSPACE 0~255
\IOPAGE 255
\STACKSPACE 256
(10000H)
\GuardStackAddr
F000H
\LastStackAddr
FFFEH(OFFSET)
\PLISTSPACE 512
(512)
\FPTOVP 1024~1279
(256)
\PAGEMAP 1280~1535
(256)
\InterfacePage 1536
\PageMapTBL 1538~1545(8)
\MISCSTATS 1546
\UFNTable 1548,1549
\DTDSpaceBase 1552~1567
(16)
\LISTPDTD 1552
\LOCKEDPAGETABLE
1648~1663(16)
\MDSTypeTable 1664~1791
(128)
\AtomHashTable 1792
\PNPSPACE 2048
\DEFSPACE 2560
\VALSPACE 3072
\SMALLPOSPSPACE 3584
\SMALLNEGSPACE 3840
\HTMAIN 4096~4223
(128)
\HTOVERFLOW 4224
\HTBIGCOUNT 4225
\HTCOLL 4352
\DISPLAYREGION 4608~4809
(202)
\FirstArrayPage 4864
\FirstMDSPage
\PNCHARSSPACE -
\PNAMESPACEEND -
\DefaultSecondArra
16384
\DefaultSecondMDS
65532

0~124	34820
126~1790	NOREF
1792~2046	32777
2048~3582	NOREF
3584~4094	47105
4096~4862	NOREF
4864~4922	0
4924~5886	NOREF
5588~10036	0
10038~10064	14338
10066~10198	MDS
10200~10234	6147
10236~10256	MDS
10258~10278	6147
10280~10384	MDS
10384~10388	14338
10390~14846	MDS
14848~14992	NOREF
14994~16382	MDS
16384~17158	0
17160~64932	NOREF
64934~65536	MDS
65538	31315

VMEM TYPE TABLE ON DLion
with 24000 pages
VMEMSIZE approximatery 16300

Spec of \ATOMSPACE

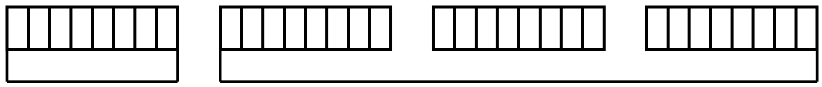
0	NIL ptr
1	NOBIND ptr
367Q	ASCII code from 0 to 47, and 58 - 255
370Q	\EVALFORM
	\GC.HANDLEOVERFLOW
	\DTEST.UFN
	\OVERFLOWMAKENUMBER
	\MAKENUMBER
	\SETGLOBAL.UFN
	\SETFVAR.UFN
	\GCMAPTABLE
400Q	\INTERPRETER
	SMALLP
	FIXP
	FLOATP
	LITATOM
	LISTP
	ARRAYP
	STRINGP
	STACKP
	CHARACTER
	VMEMPAGEP
	STREAM
	BITMAP
	COMPILED-CLOSURE
	ONED-ARRAY
	TWOD-ARRAY
	GENERAL-ARRAY
	\PTRHUNK2
	\PTRHUNK4
	\PTRHUNK5
	\PTRHUNK6
	\PTRHUNK8
	\PTRHUNK10
	\PTRHUNK12

\PTRHUNK16
\PTRHUNK24
\PTRHUNK32
\PTRHUNK42
\PTRHUNK64
\UNBOXEDHUNK1

* CELL[redacted]

MSB

Interlisp-D

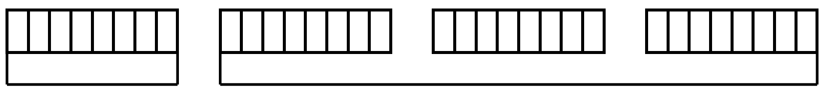


CDR-code
(It may used as
the CELL offset in
the CONSPAGE)

It pointes to the CELL by WORD addressing

MSB

Interlisp-68k



CDR-code
(It may used as
the CELL offset in
the CONSPAGE)

It pointes to the CELL by BYTE addressing
It causes we must use 16 Mbyte
address space,not 32 M byte at maximam

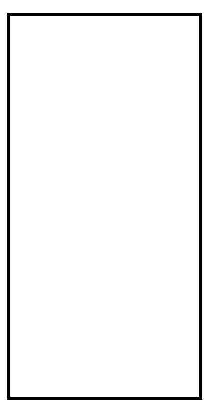
* CONSPAGE

Word offset address

0x00

Interlisp-D

128 CELL size



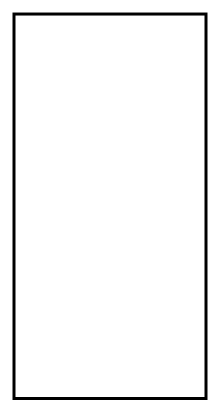
0x0FF

Byte offset address

0x00

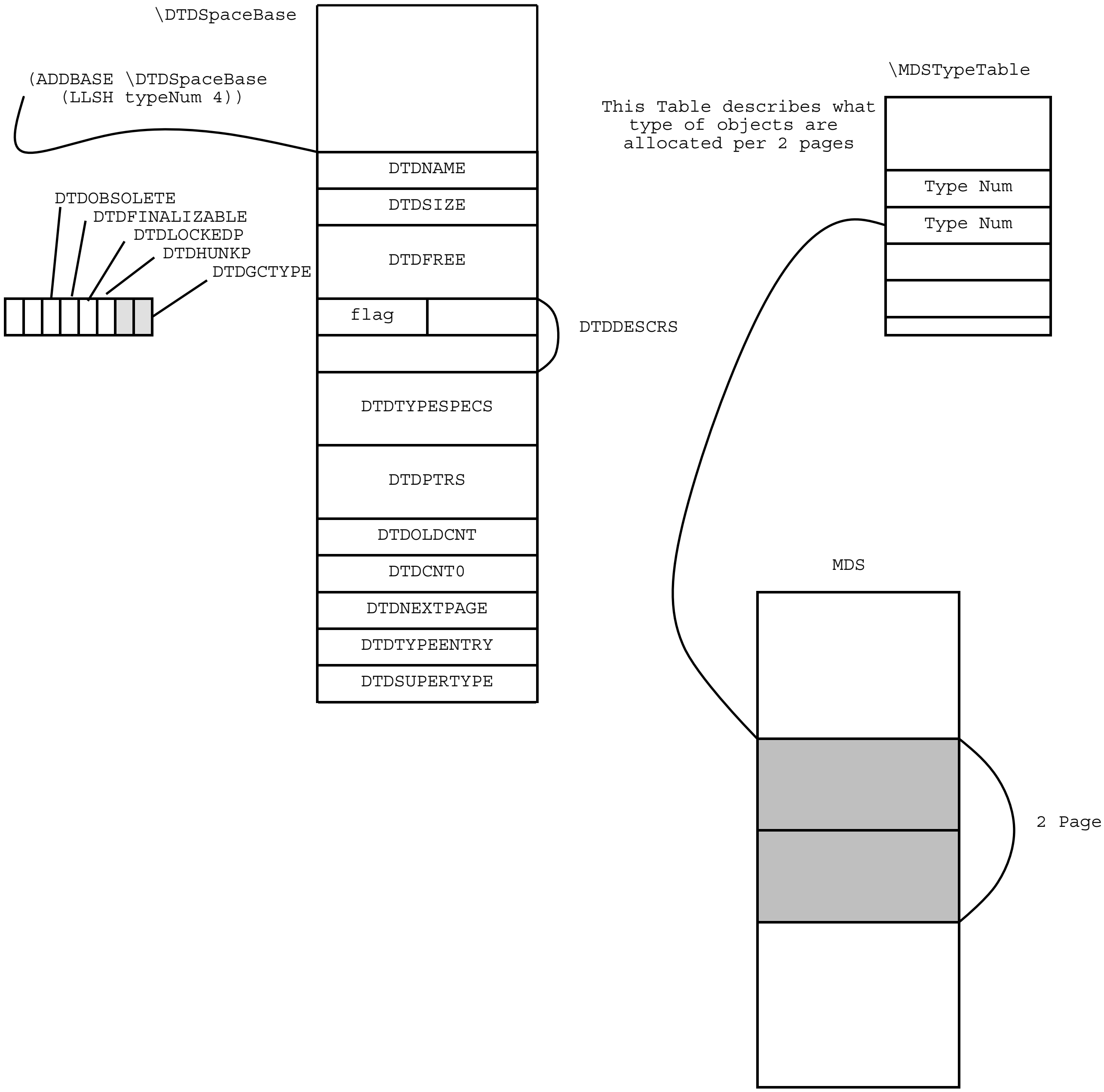
Interlisp-68k

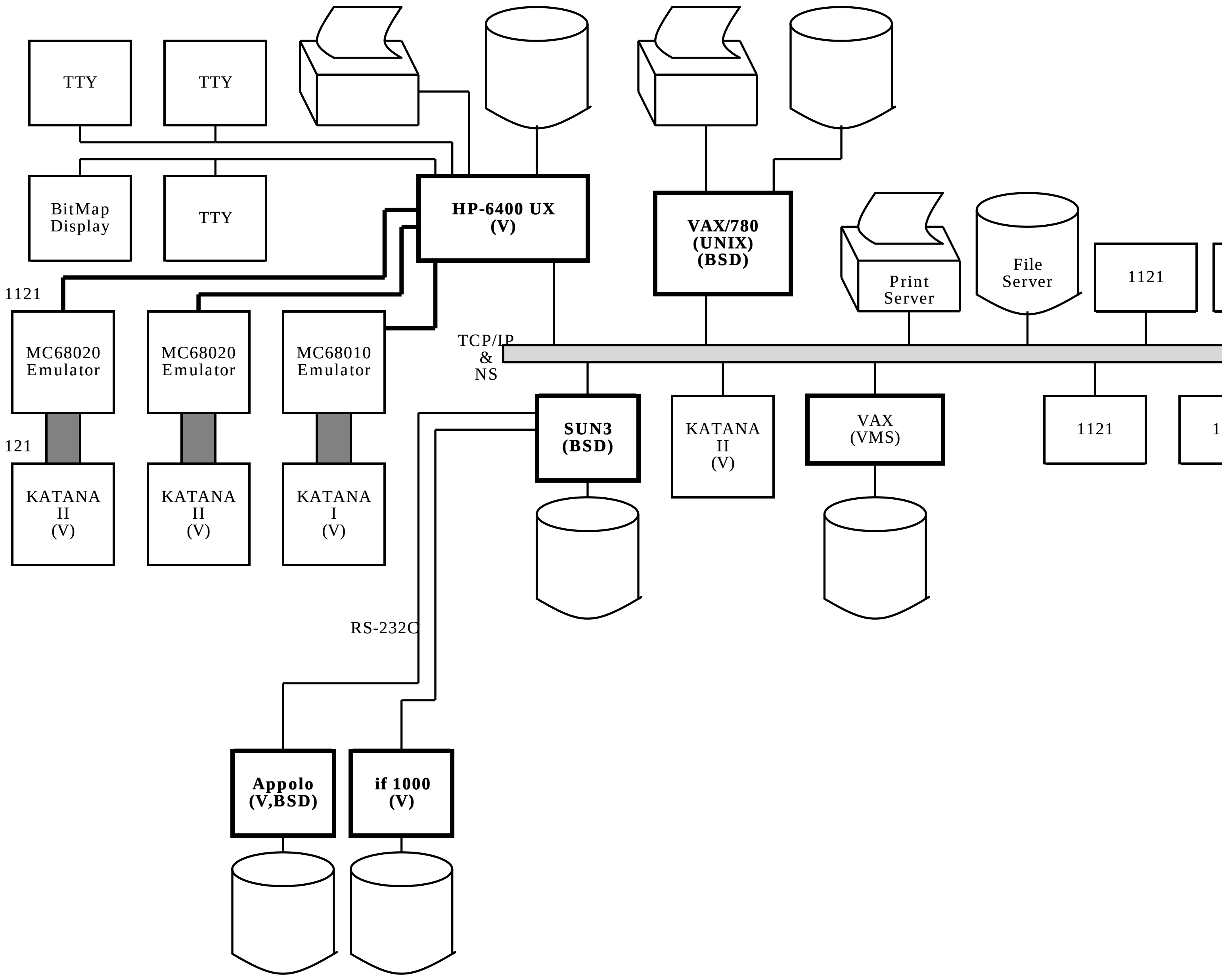
128 CELL size



0x1FF

: 1 Cell 32 bit width, and !CONSPAGE includes 128 Cells

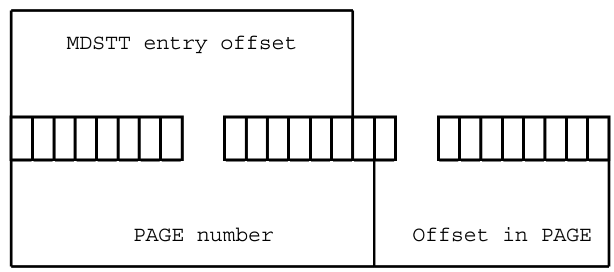

\DTDSpaceBase
(ADDBASE \DTDSpaceBase
(LLSH typeNum 4))
DTDNAME
DTDSIZE
DTDOBSOLETE
DTDFINALIZABLE
DTDLOCKEDP
DTDHUNKP
DTDGCTYPE
DTDFREE
flag
DTDDESCRS
DTDTYPESPECS
DTDPTRS
DTDOLDCNT
DTDCNT0
DTDNEXTPAGE
DTDTYPEENTRY
DTDSUPERTYPE
\MDSTypeTable
This Table describes what
type of objects are
allocated per 2 pages
Type Num
Type Num
MDS
2 Page



TTY
TTY
BitMap Display
TTY
HP-6400 UX (V)
VAX/780 (UNIX) (BSD)
Print Server
File Server
1121
1121
MC68020 Emulator
MC68020 Emulator
MC68010 Emulator
TCP/IP & NS
121
KATANA II (V)
KATANA II (V)
KATANA I (V)
SUN3 (BSD)
KATANA II (V)
VAX (VMS)
1121
RS-232C
Appolo (V,BSD)
if 1000 (V)

Data Type recognition on KATANA

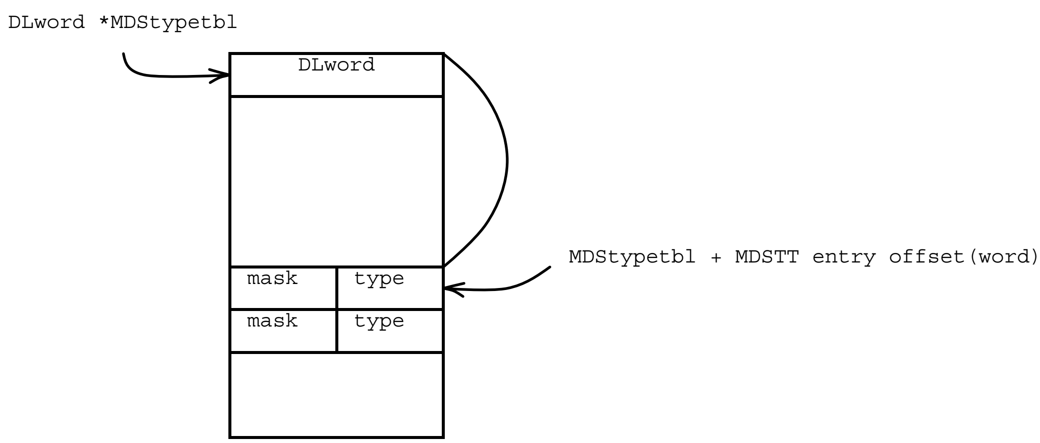
by Takeshi Shimizu



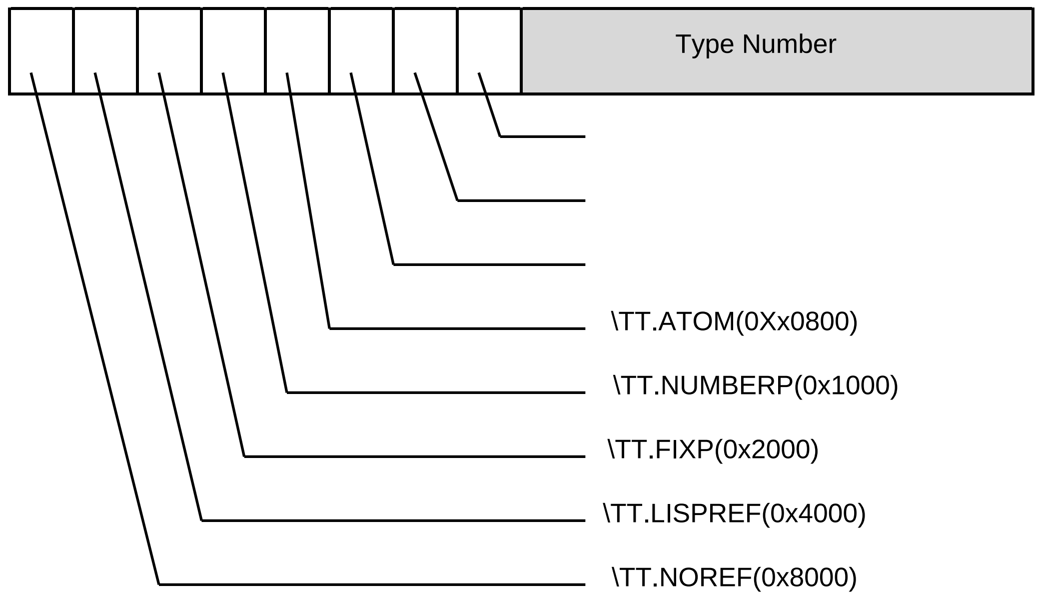
1 Page = 512 byte = 9 bit = 0x1FF

1 MDS entruy explains the type of 2 pages -- 10 bit

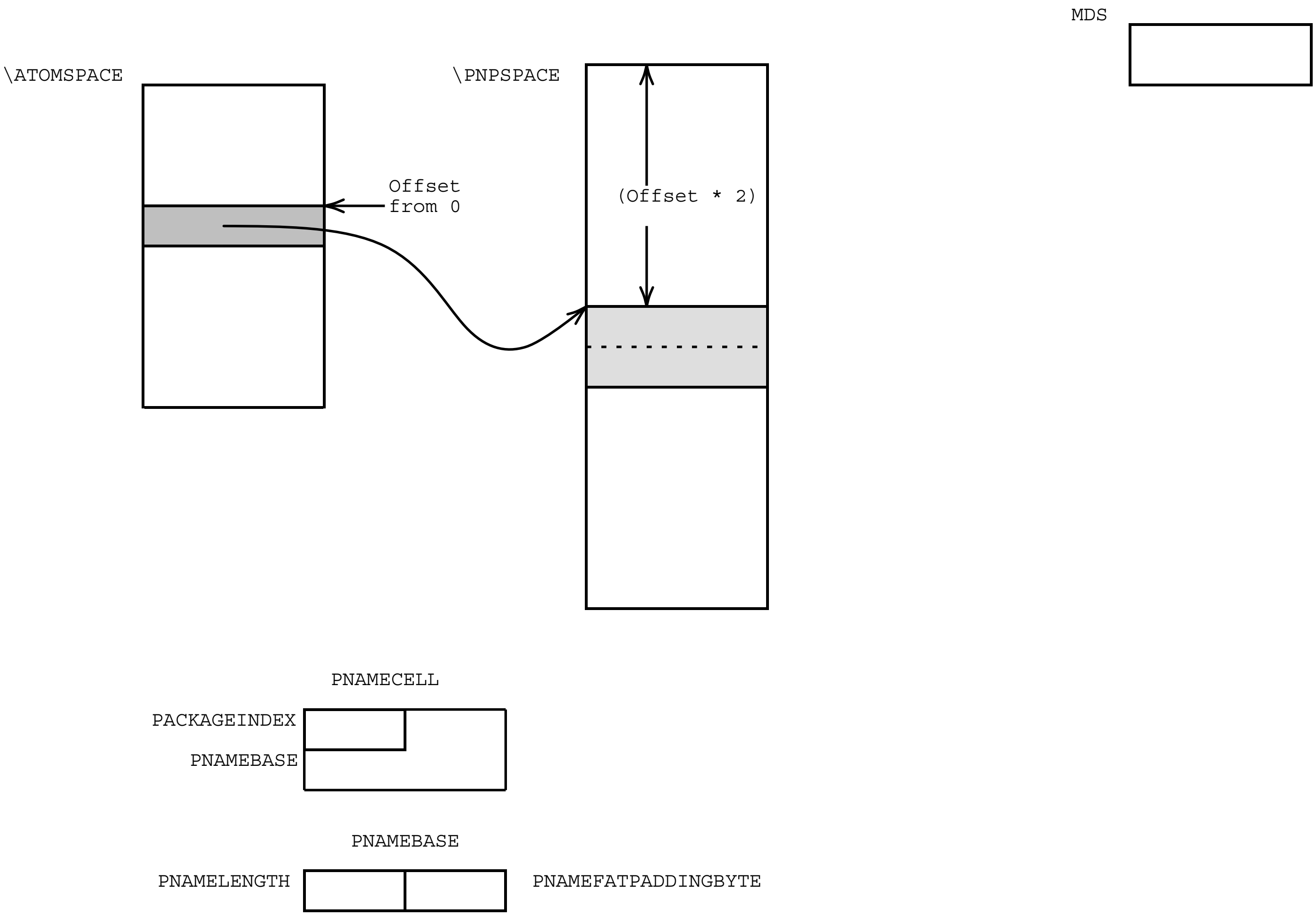
Method of accessing to MDSTT



MDSTtpeTable Entry

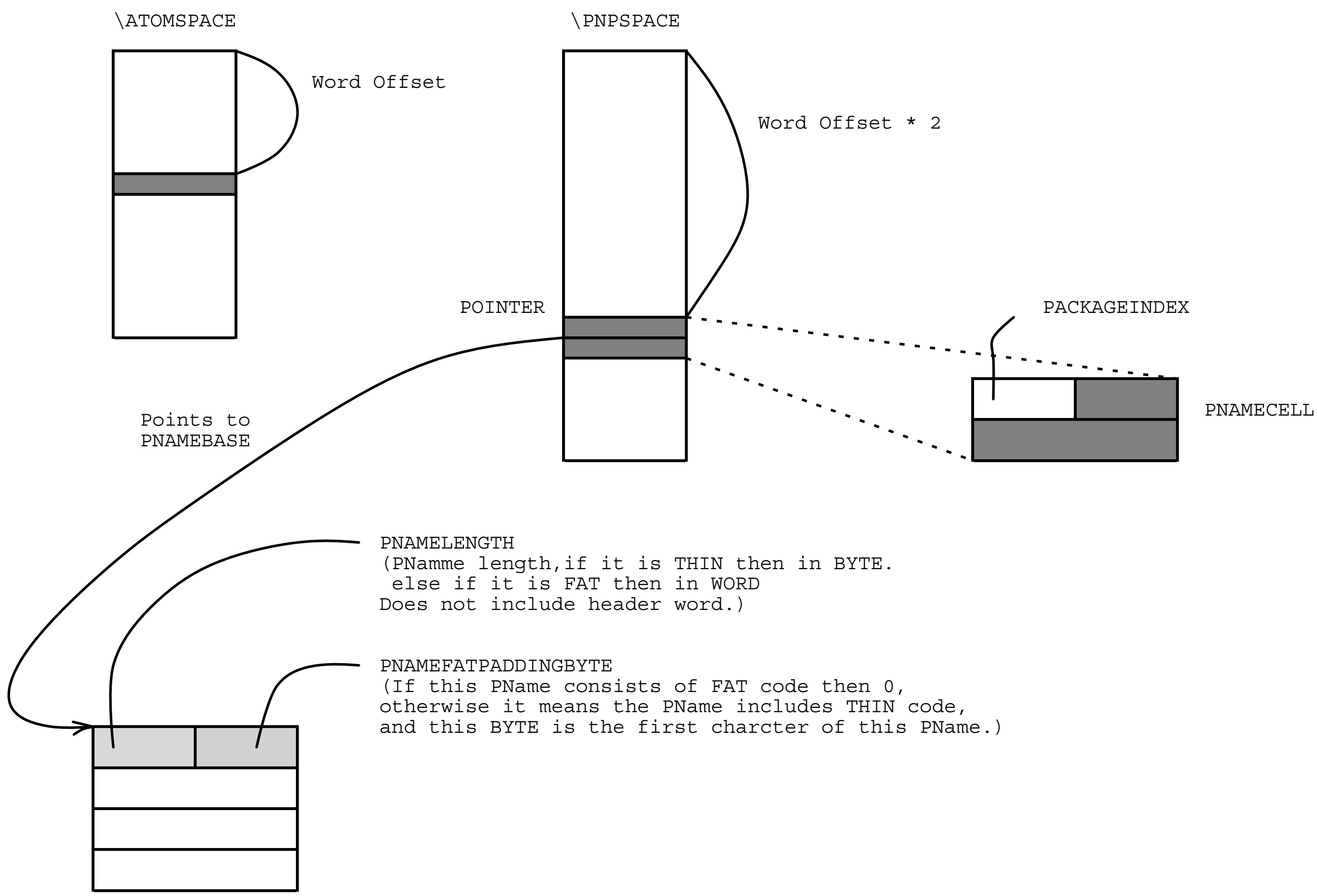


How to Access PName



How to Access PName

by Takeshi Shimizuu



Example: NIL

Its index number is 0.
And,Its Length is 3 bytes.

3	N
I	L

Example

Example: [redacted]

Its index number is X.
And,Its Length is 3 wordes.

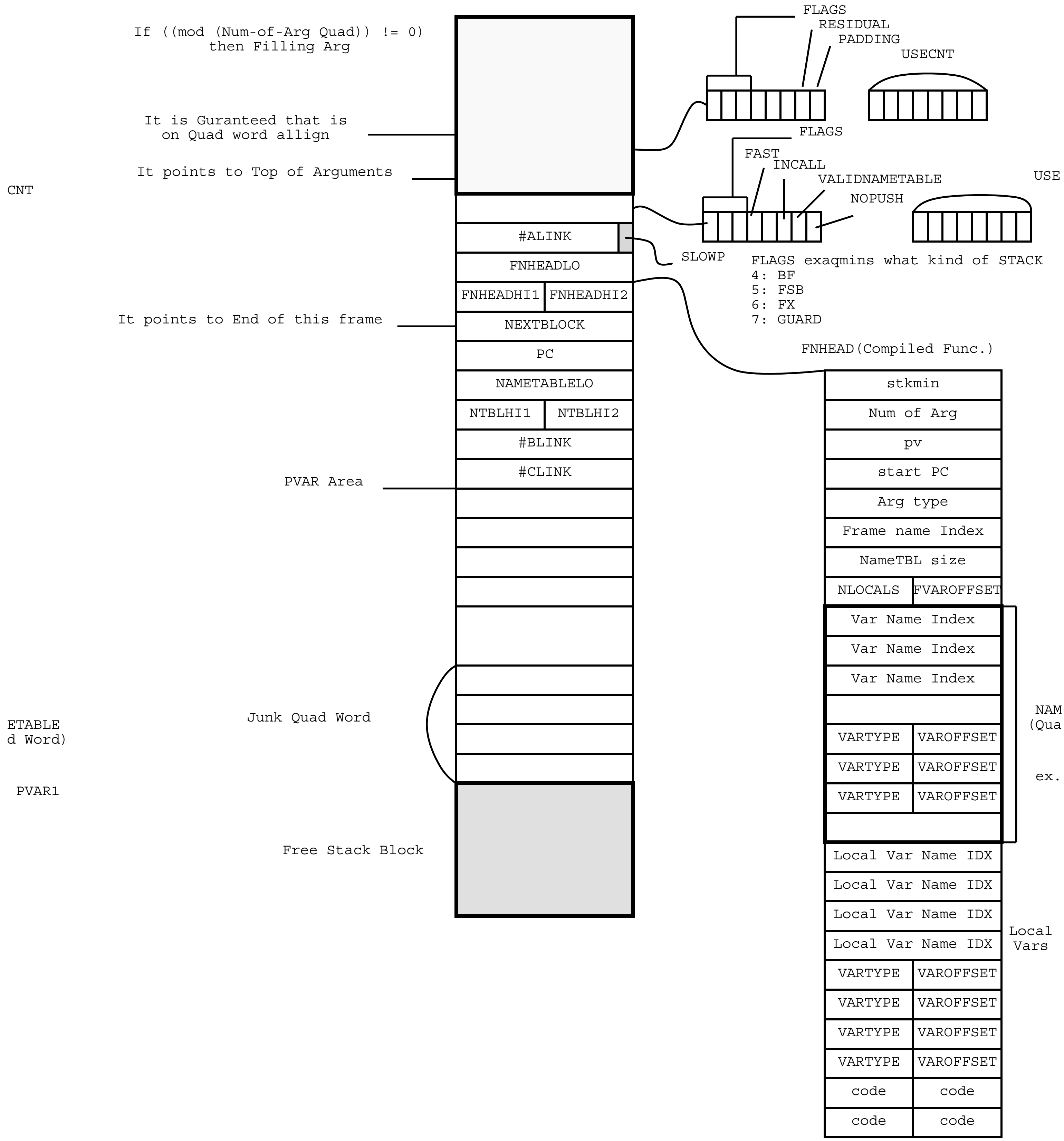
3	0
[redacted]	
[redacted]	
[redacted]	

Lyric Micro Code [redacted]

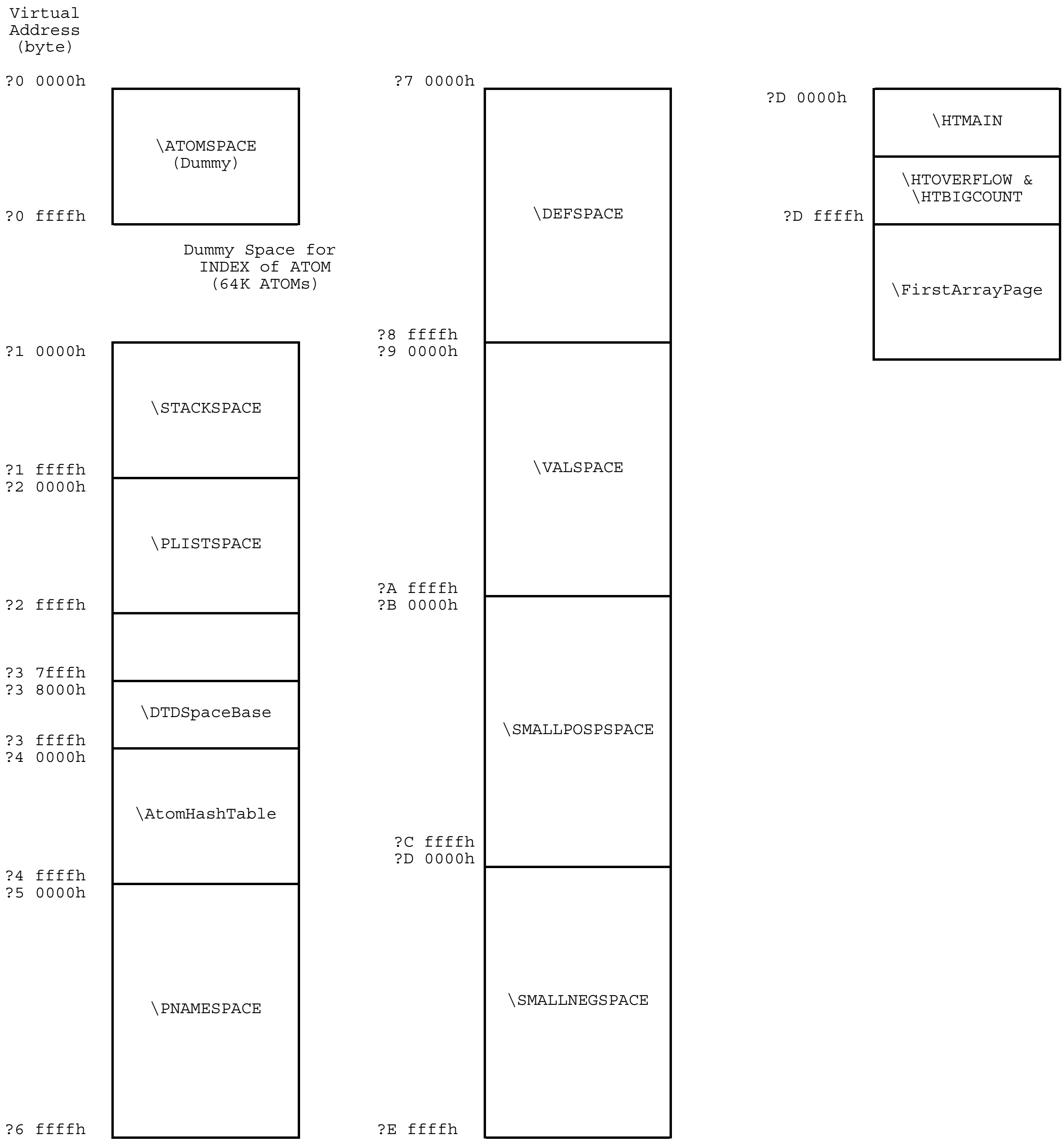
Combination TEST
start

AME	11/E	12/B	12/M	12/E	1/B	1/M	1/E	2/B	2/M	2/E	3/B	3/
Shimizu	Investigations on Initial Memory Allocations in LISP		Coding & Test C functions (Memory Allocations)			Combination TEST			Integration Test			
Inagaki	Investigations on List handling & Data Type checking uCODE 12 Entries		Investigations on Function call & Stack manupilations uCODE 15 Entries		Investigations on Other Fundamentals uCODE 22 entries		Investigation on Garbage Collection uCODE 6 Entries		Investigations on Another rest uCODE			
Hayata (Sato) t	Documentation Investigations on Variable manipulation uCODE 18 entries			Investigations on Jump instructions uCODE 9 Entries		Investigations on Low Level addressing uCODE 14 Entries					Integration Tes	
			Coding in C & Test(with Sato)									
Mitani	Investigations on Data Type allocations in LISP					Combination TEST			Integration Test			
??	Investigations what will be needed when prototype will work (LISP..e.g. ARRAY)		Documentation									

Stack Contents (if Argments gives as 2) revised 6-Feb-87

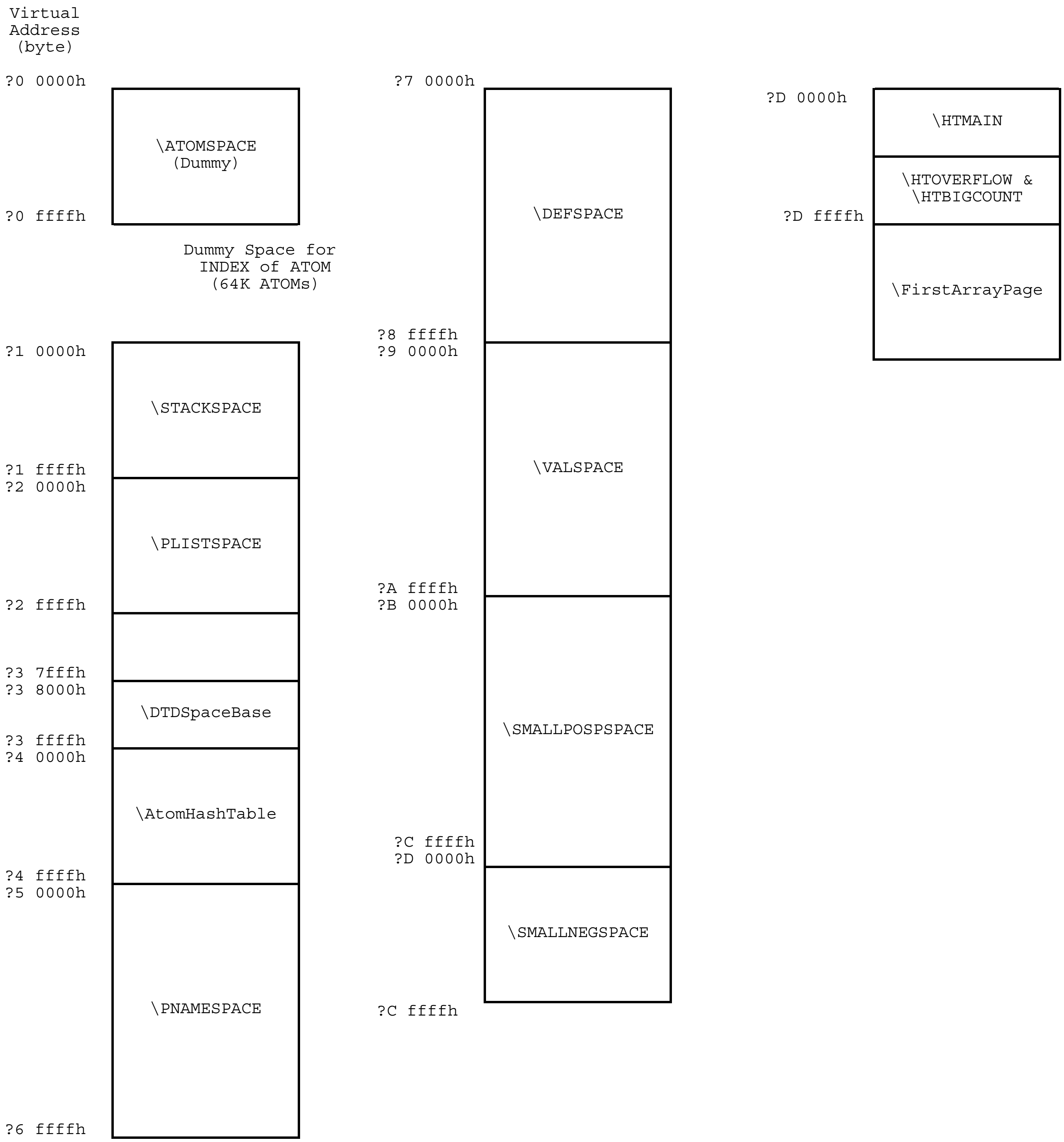


Memory Map(for TEST on SUN-3)



(ADDBASE (LLSH ATOMINEX 1) \PNAMESPACE)

Memory Map(for TEST on SUN-3)



(ADDBASE (LLSH ATOMINEX 1) \PNAMESPACE)

\ADVANCE_ARRAY_SEG(NXTPAGE)

Called when the first 8mb are exhausted, and we want to switch array space into the next area, starting with page NXTPAGE -- have to first clean up what's left in the old area

NCELLSLEFT=
(*NxtArrayPage_word - POINTER_PAGE(ArrayFrLst)-1) * CELLSPER_PAGE +
(CELLSPER_PAGE - POINTER_CELLINPAGE(\ArrayFrLst)

NCELLSLEFT >= \MinArrayBlockSize (4)

\\MERGEBACKWARD(\MAKEFREEARRAYBLOCK
(\ArrayFrLst,NCELLSLEFT) ; Make the rest of the already allocated array space into a small block

\ArrayFrLst2=\LeastMDSPage<<8

\ArrayFrLst2=\ArrayFrLst

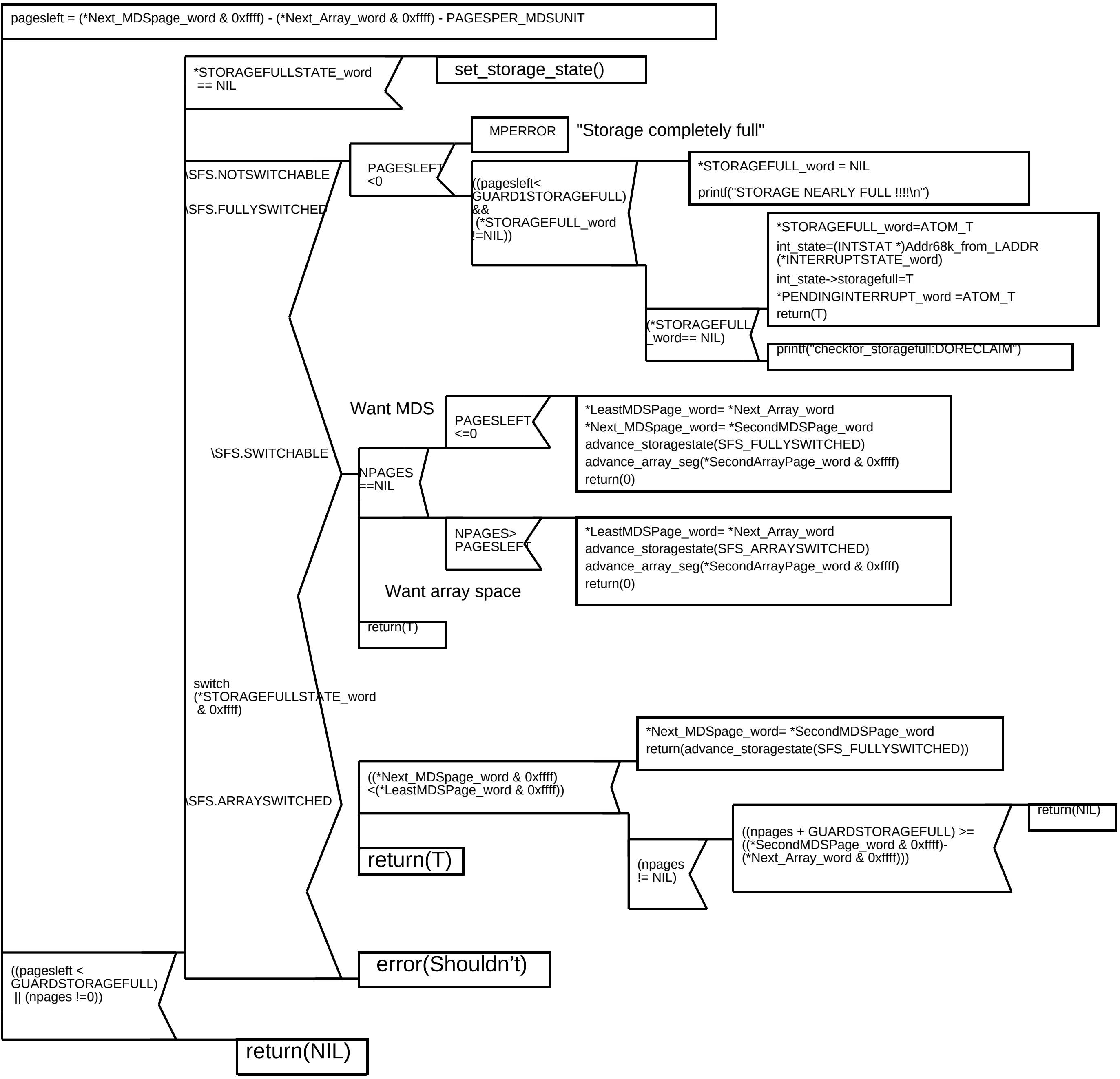
\NxtArrayPage=NXTPAGE ; Return code to tell \ALLOCBLOCK.NEW to notice the new arrangement

\ArrayFrLst=NXTPAGE<<8

\ARRAYSPACE2=\ArrayFrLst

return(0)

checkfor_storagefull(npages)



[redacted]
[redacted]CONSPAGE[redacted]

cons (cons_car , cons_cdr)

```
gclookup(ADDREF, cons_car);
gclookup(ADDREF, cons_cdr);
```

[redacted]Ref. cnt[redacted]

```
new_page = ListpDTD->dtd_nextpage;
new_conspage =
 (struct conspage *)Addr68k_from_LPAGE(new_page);
new_cell = GetNewCell_68k(new_conspage);
new_conspage->count --;
new_conspage->next_cell = new_cell->cdr_code ;
```

```
new_cell->car_field = cons_car ;
new_cell->cdr_code = CDR_NIL ;
```

[redacted]

```
((ListpDTD->dtd_nextpage != 0)
&&
(GetCONSCount(ListpDTD->dtd_nextpage) > 0))
```

ListpDTD[redacted]CONSPAGE
[redacted]
1[redacted]Free[redacted]

```
ListpDTD->dtd_cnt0++
```

[redacted]

CDRing NIL

```
(cons_cdr
== NIL_PTR)
```

[redacted]CONSPAGE[redacted]

```
new_conspage=next_conspage()
```

[redacted]CONSPAGE[redacted]

```
new_cell = GetNewCell_68k( new_conspage ) ;
new_conspage->count --;
new_conspage->next_cell = new_cell->cdr_code ;
```

```
new_cell->car_field = cons_car ;
new_cell->cdr_code = CDR_NIL ;
```

[redacted]

```
ListpDTD->dtd_oldcnt++;
```

[redacted]

CDR[redacted]CONSPAGE
[redacted]

```
new_page
 = POINTER_PAGE(cons_cdr);
```

CDR[redacted]CONSPAGE[redacted]

On Page

```
new_conspage=
 (struct conspage *)Addr68k_from_LPAGE(new_page);
new_cell = GetNewCell_68k( new_conspage ) ;
new_conspage->count --;
new_conspage->next_cell = new_cell->cdr_code ;
```

[redacted]

```
(listp(cons_cdr)
  &&
(GetCONSCount(new_page ) > 0 ))
```

CDR[redacted]
1[redacted]Free[redacted]

```
new_cell->car_field = cons_car ;
new_cell->cdr_code = CDR_ONPAGE | (LOLOC(cons_cdr)>>1);
```

```
ListpDTD->dtd_cnt0++
```

[redacted]

[redacted]CONSPAGE[redacted]

```
new_conspage=next_conspage()
```

[redacted]CONSPAGE[redacted]2[redacted]

Diff. Page

```
temp_cell = GetNewCell_68k( new_conspage ) ;
new_conspage->next_cell = temp_cell->cdr_code ;
new_cell = GetNewCell_68k( new_conspage ) ;
new_conspage->next_cell = new_cell->cdr_code ;
new_conspage->count -= 2;
```

CDR[redacted]

```
temp_cell->car_field = cons_cdr ;
temp_cell->cdr_code = 0 ;
```

CAR[redacted]

```
new_cell->car_field = cons_car ;
new_cell->cdr_code
=  (LOLOC(LADDR_from_68k(temp_cell)) >> 1) ;
```

```
ListpDTD->dtd_oldcnt++;
```

[redacted]

```
ret_Laddr = LADDR_from_68k( new_cell )
```

```
gclookup ( DELREF , ret_Laddr )
```

```
return ( ret_Laddr)
```

OP_contextswitch

PushCStack

contextswitch(TopOfStack &0xffff)

CurrentFX =STK_OFFSET | Addr68k_from_LADDR(*(InterfacePage + fxnum))

*(InterfacePage + fxnum) = LOLOC(LADDR_from_68k(CurrentFX))

Midpunt

contextswitch(fxnum)

fxnum ==FAULTFIXP

CurrentFX->incall=T

CurrentFX->nopush=T

CurrentFX->pc =PC-FuncObj + 1

CurrentFX->nextblock = ++CurrentStackPTR

*CurrentStackPTR = FSB_MARK

*(CurrentStackPTR+1) = StkLim0-CurrentStackPTR+STKLIMOFFSET

CurrentStackPTR++

Midpunt

RTN2 (*next68k !=FSB_MARK)

error("OP_return : MP9316")

freeptr= next68k

*freeptr==FSB_MARK

EndSTKP = freeptr=freeptr +*(freeptr+1)

StkLim0=EndSTKP-7

next68k >=StkLim0

Midpunt

goto RTN2

PVar =returnFX+FRAMESIZE

returnFX ->incall

returnFX->incall=0

CurrentStackPTR = next68k-2

TopOfStack = *((LispPTR *)CurrentStackPTR)

CurrentStackPTR--

PC = returnFX->pc

OP_apply()

returnFX->nopush=0

CurrentStackPTR = next68k-2

TopOfStack = *((LispPTR *)CurrentStackPTR)

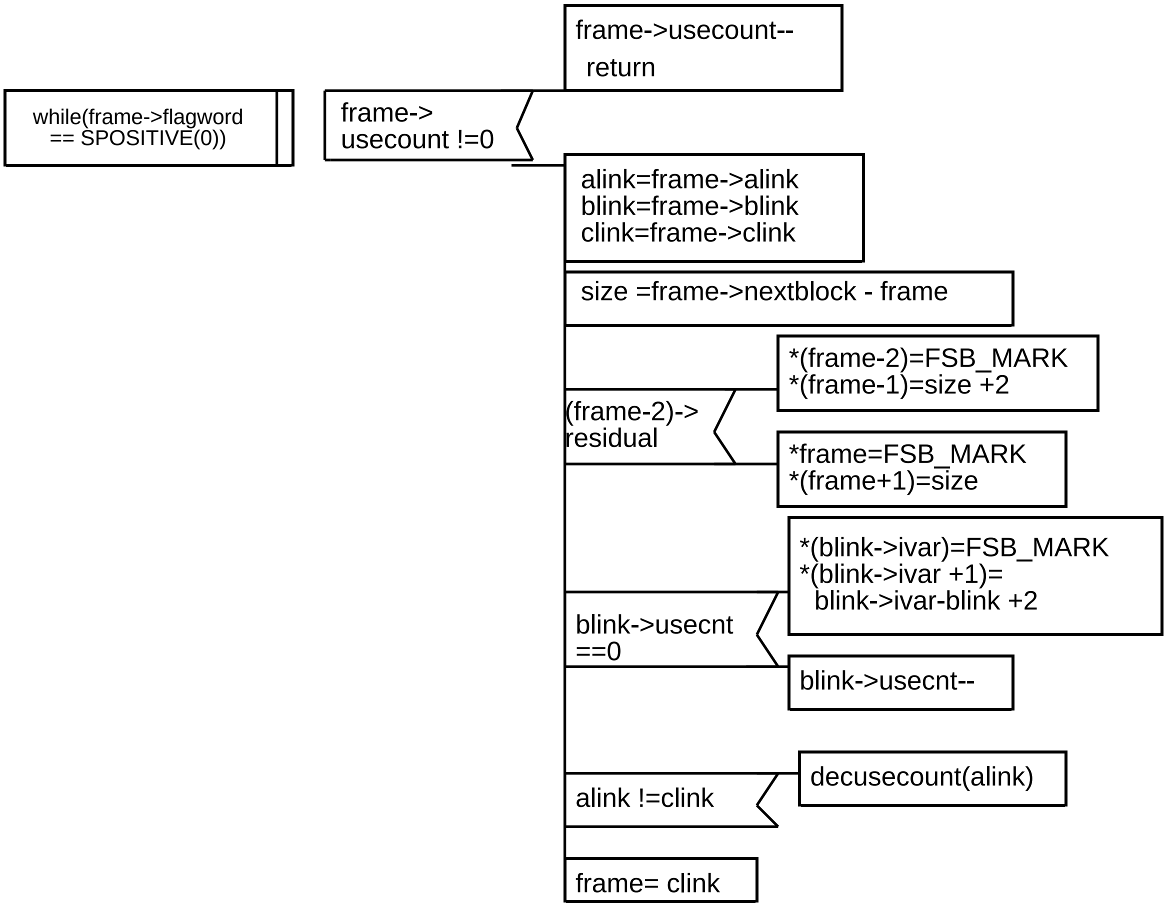
returnFX ->nopush

CurrentStackPTR--

CurrentStackPTR = next68k-1

FastRetCALL

DECUSECOUNT(FRAME)



SETACLINK(fx,val) { if(!(fx->alink&1))fx->blink=fx-DLWORDSPER_CELL;
fx->clink=val+FRAMESIZE;
fx->alink=fx->clink+1

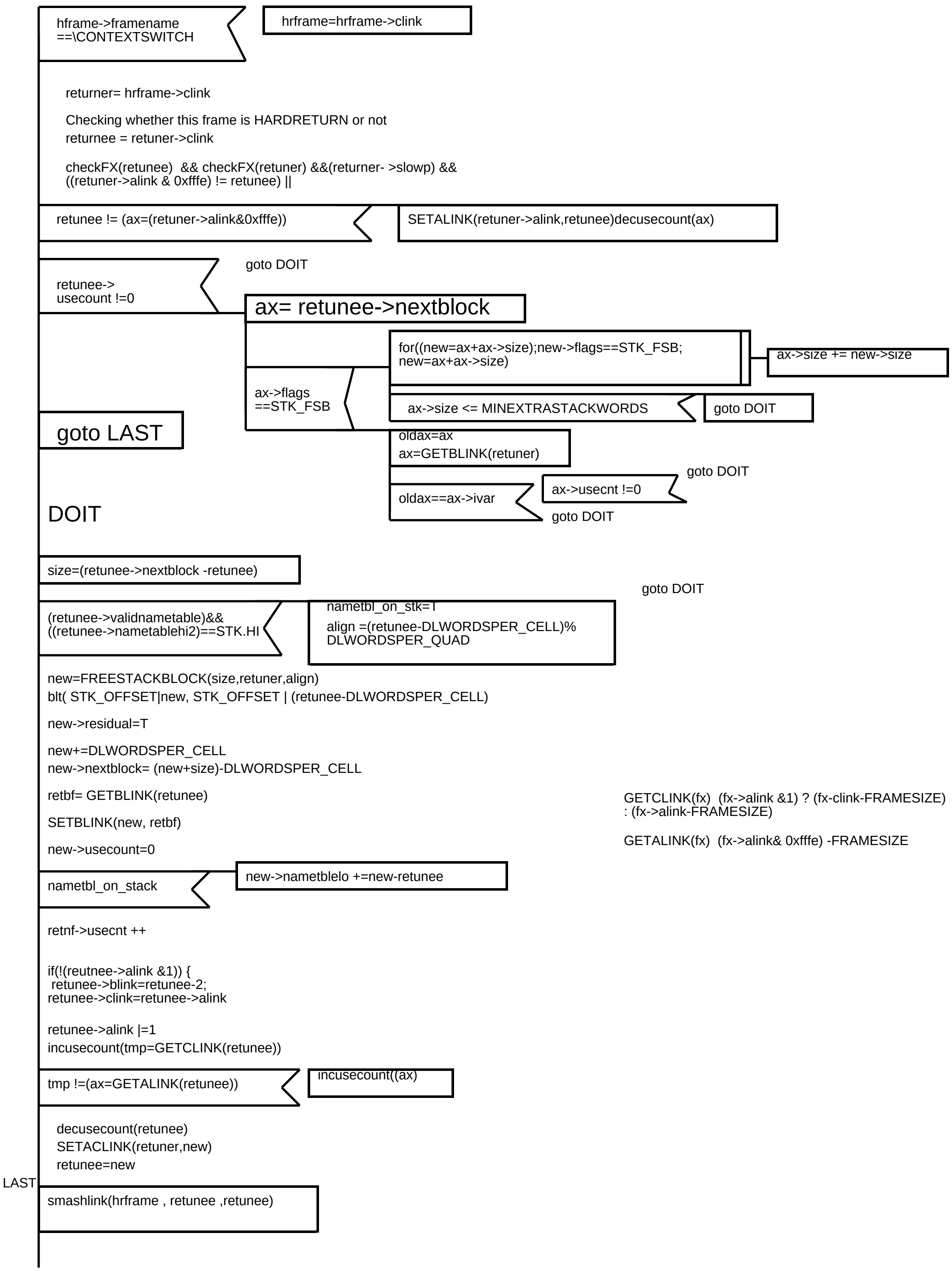
SETCLINK(fx,val) { fx->clink=val+FRAMESIZE;
if(!(fx->alink & 1)) { fx->blink=fx-DLWORDSPER_CELL;
fx->alink |= 1;}

GETBLINK(fx)
(fx->alink &1) ? fx->blink
: fx-DLWORDSPER_CELL

SETALINK(fx,val) {if(!(fx->alink&1)) {
fx->blink=fx - DLWORDSPER_CELL;
fx->clink=fx->alink}
fx->alink=val+FRAMESIZE+1;

SETBLINK(fx,val) { fx->blink=val;
if(!(fx->alink & 1))
{ fx->clink= fx->alink; fx->alink |= 1;}}

\dohardreturn1(hrframe)



GETCLINK(fx) (fx->alink &1) ? (fx-clink-FRAMESIZE)
: (fx->alink-FRAMESIZE)

GETALINK(fx) (fx->alink& 0xfffe) -FRAMESIZE

Replanning 24 March 1987 By

Takeshi Shimizu

6

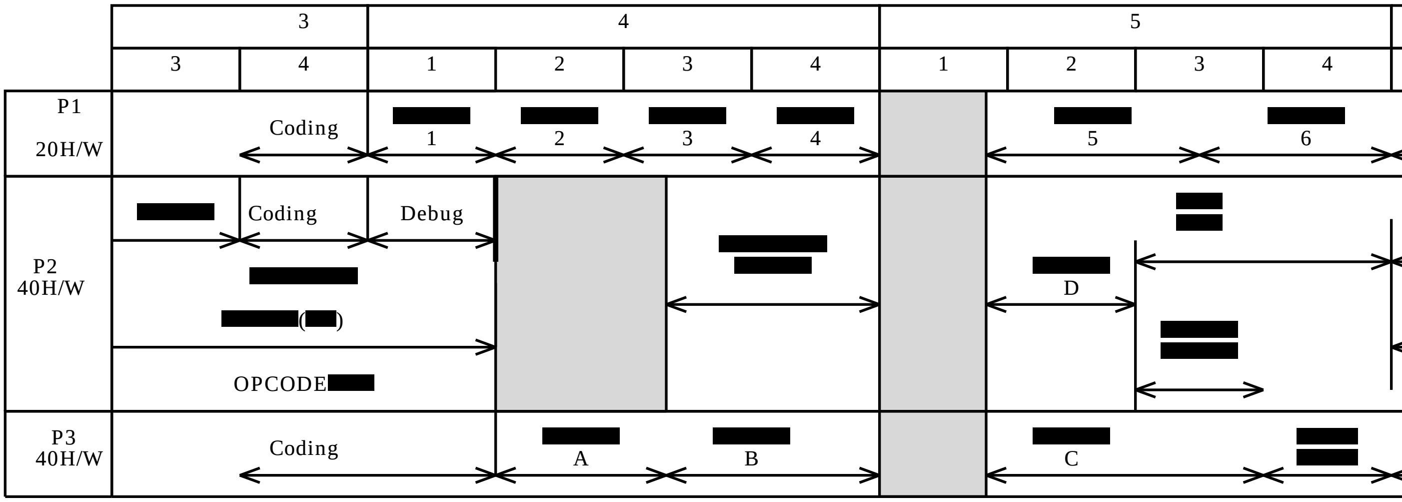
3

[redacted]

[redacted]

[redacted]

[redacted]



[redacted]1 = { vars.c,vars2.c,binds.c,jump.c,arith.c }

[redacted]2 = { lowlevel.c }

[redacted]3 = { vars3.c }

[redacted]4 = { gvar2.c }

[redacted]5 = { fvar.c }

[redacted]6 = { gc.c }

[redacted] = { initdatatype.c , makeatom.c , array.c }

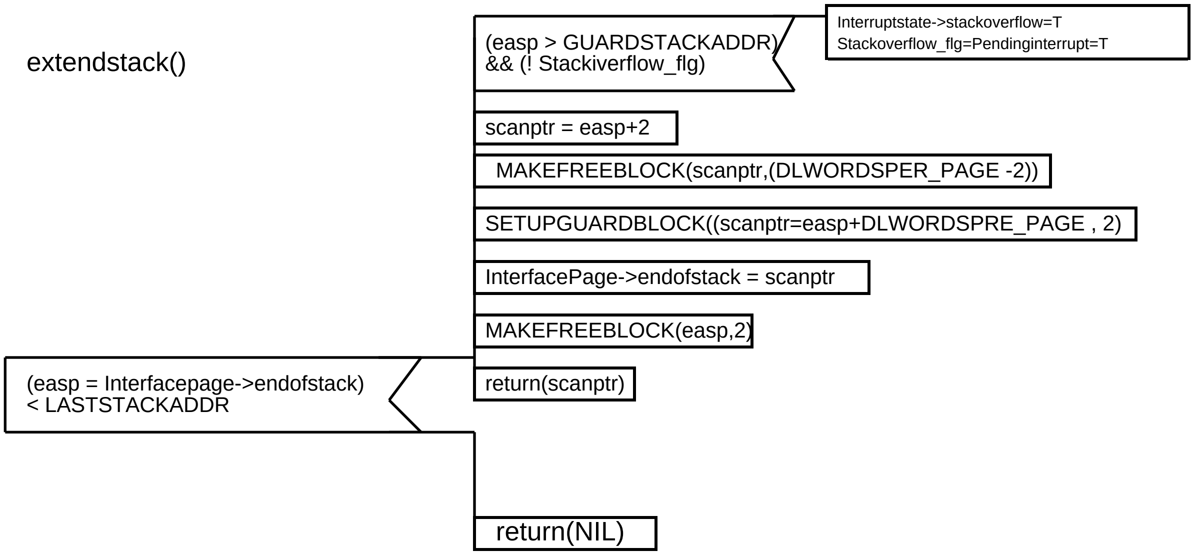
[redacted]A = { typeof.c, constants.c, OP_eq, OP_copy , OP_atomcellN }

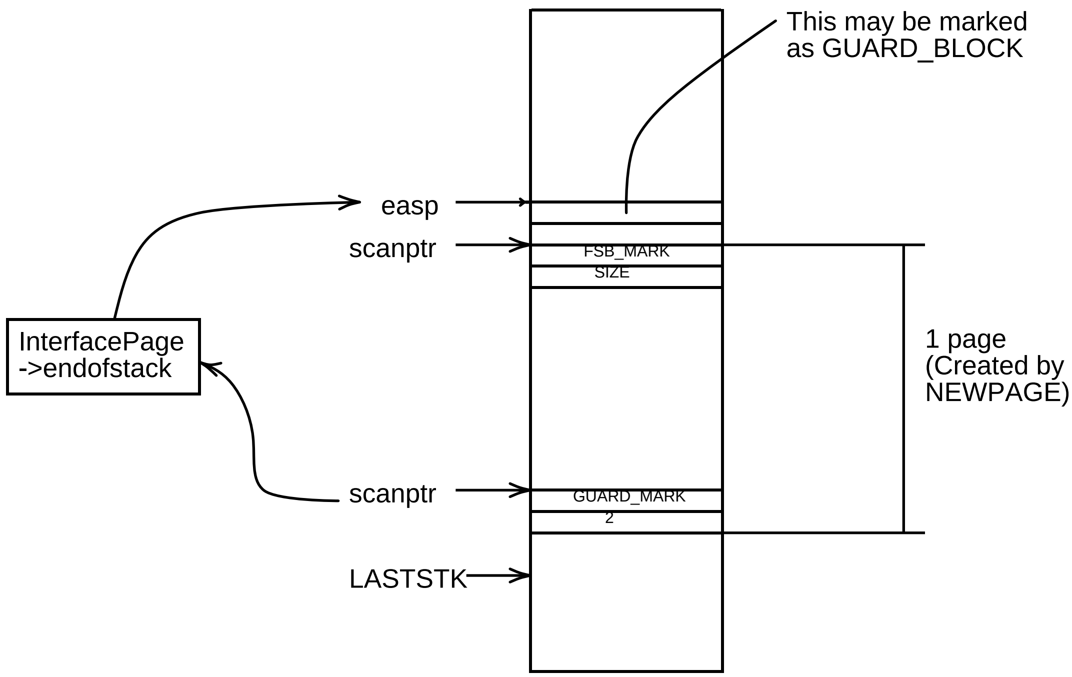
[redacted]B = { conspage.c , stack.c }

[redacted]C = { car-cdr.c }

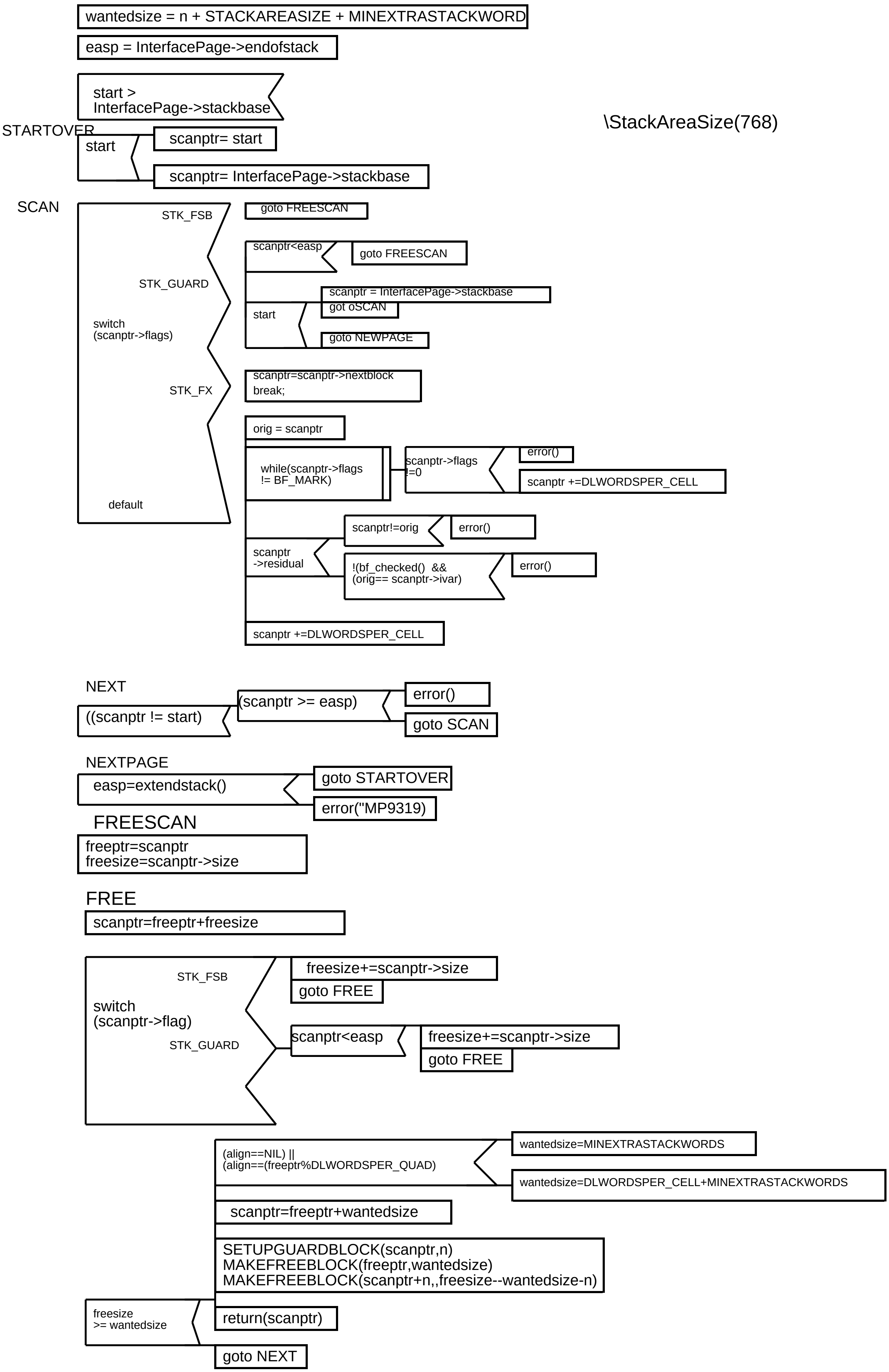
[redacted]D = { funcall.c }

[redacted] = READ,PRINT,LOADER[redacted] extendstack()



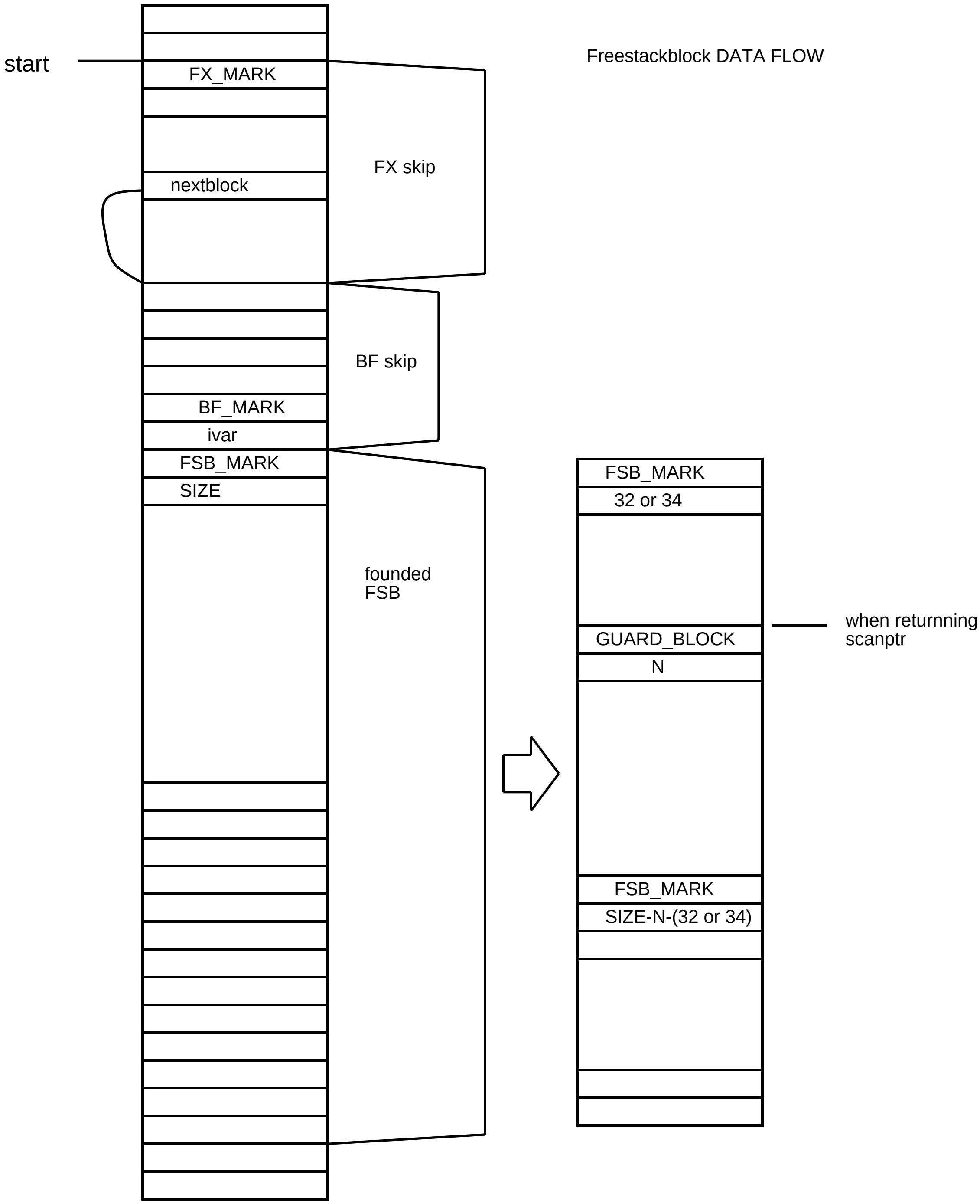


FREESTACKBLOCK(n,start,align)



\StackAreaSize(768)

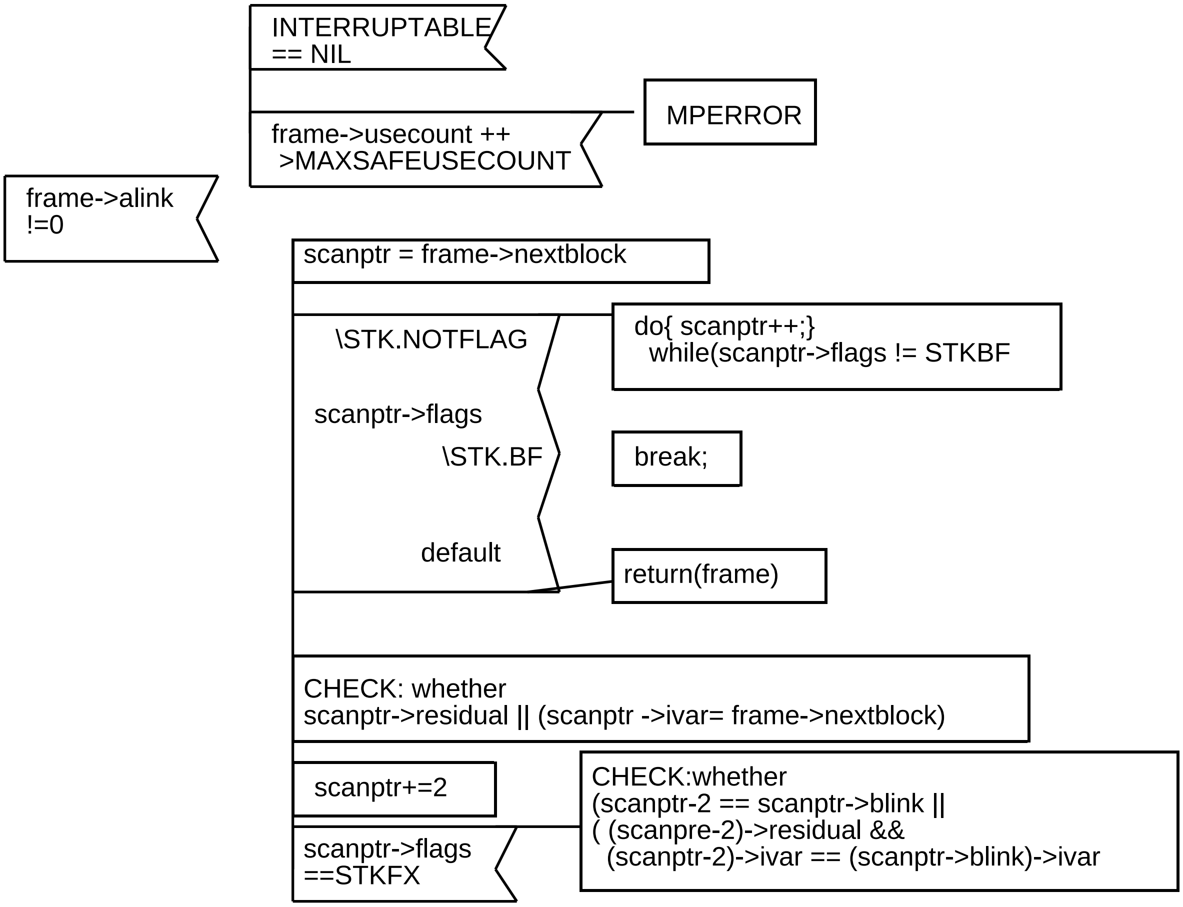
\MinExtraStackWords(32)

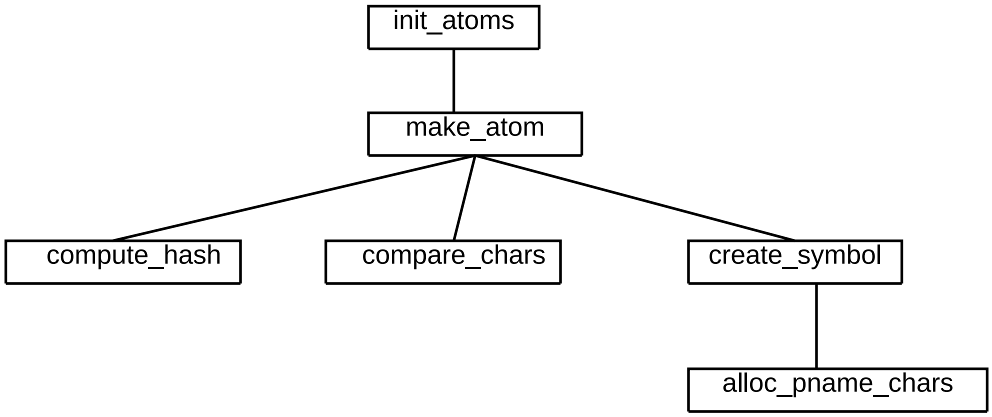


Freestackblock DATA FLOW
start
FX_MARK
nextblock
FX skip
BF skip
BF_MARK
ivar
FSB_MARK
SIZE
founded
FSB
FSB_MARK
32 or 34
GUARD_BLOCK
N
when returnning
scanptr
FSB_MARK
SIZE-N-(32 or 34)

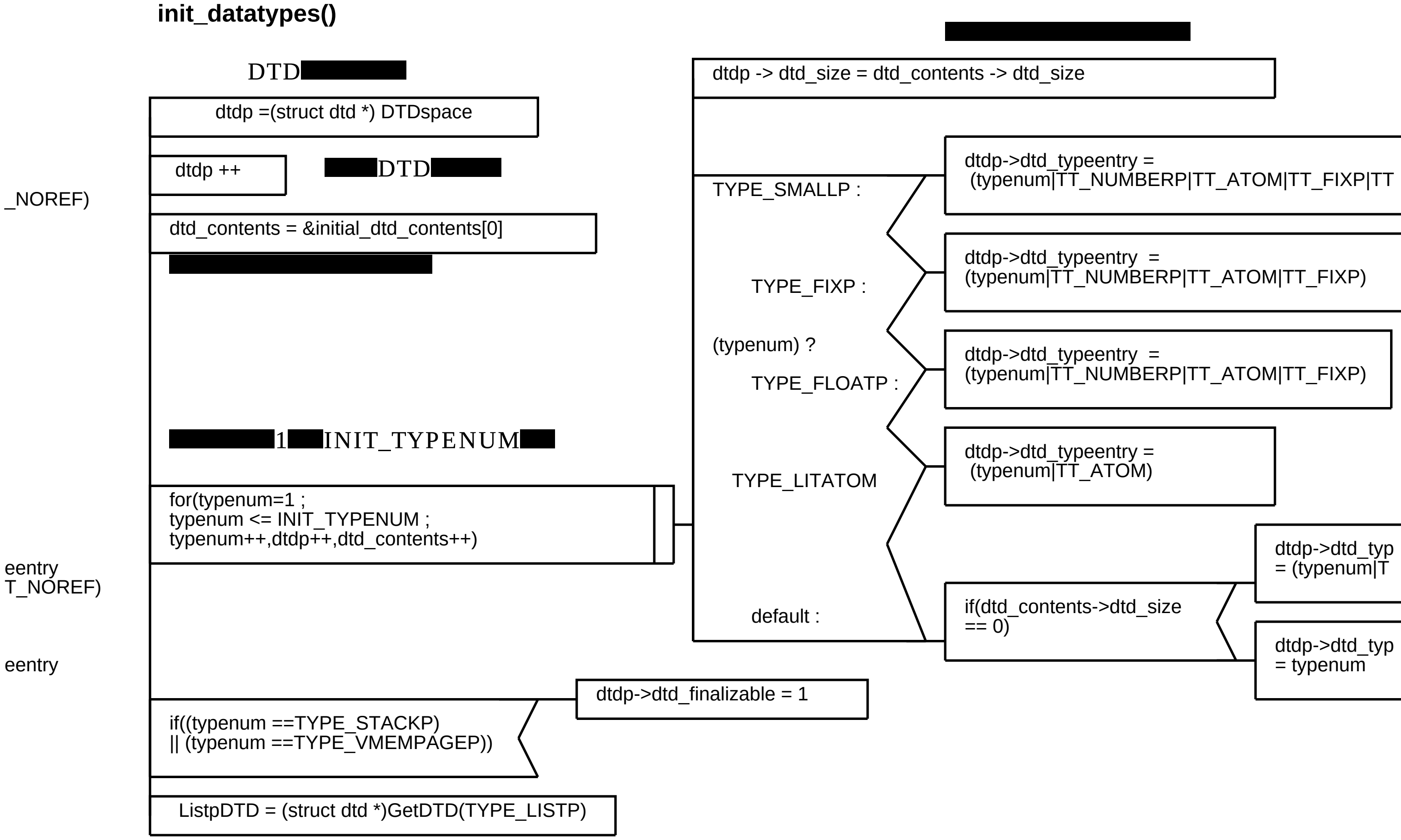
\INCUSECOUNT (FRAME)

struct frameex1 *frame

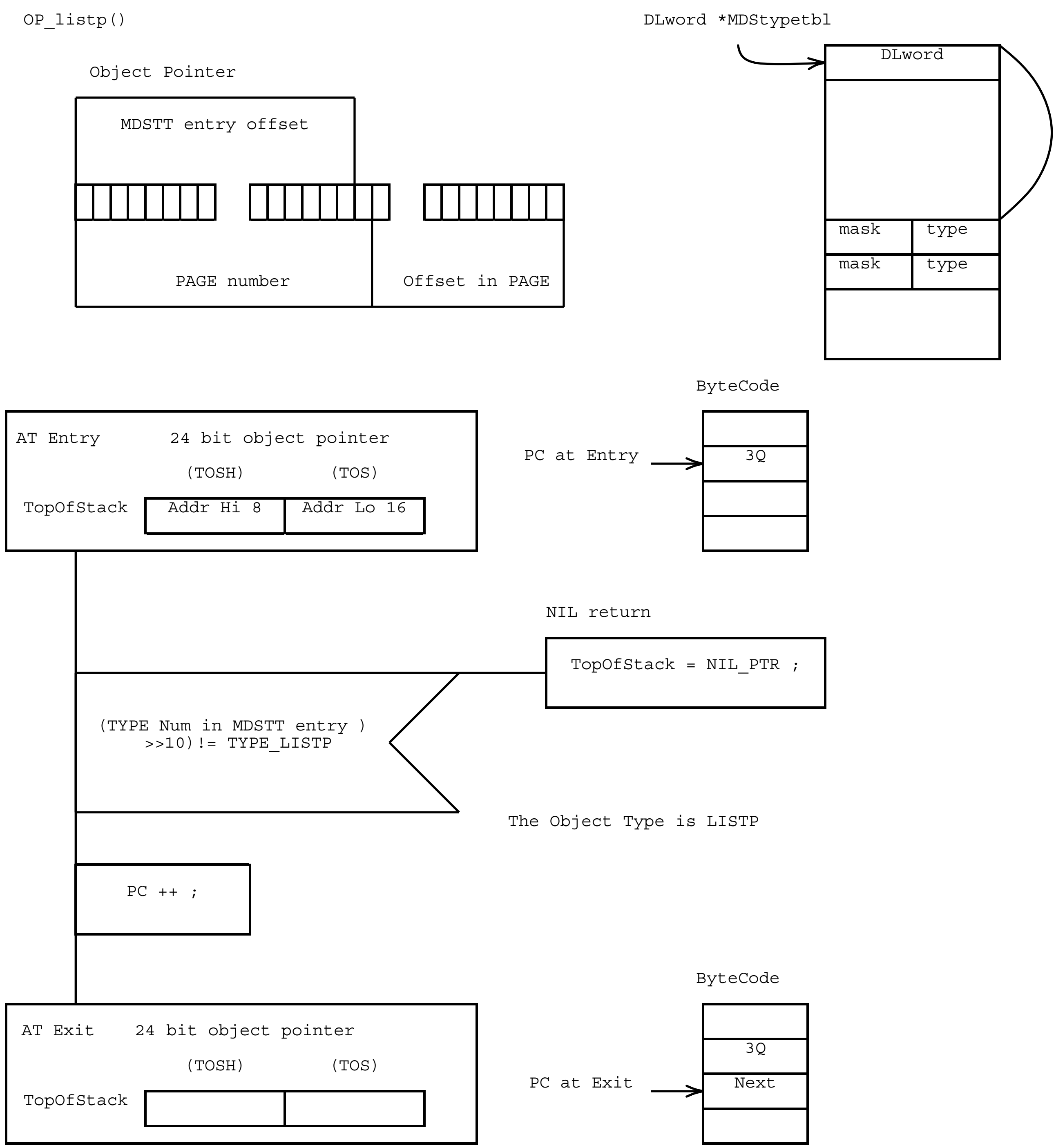



init_atoms
make_atom
compute_hash
compare_chars
create_symbol
alloc_pname_chars

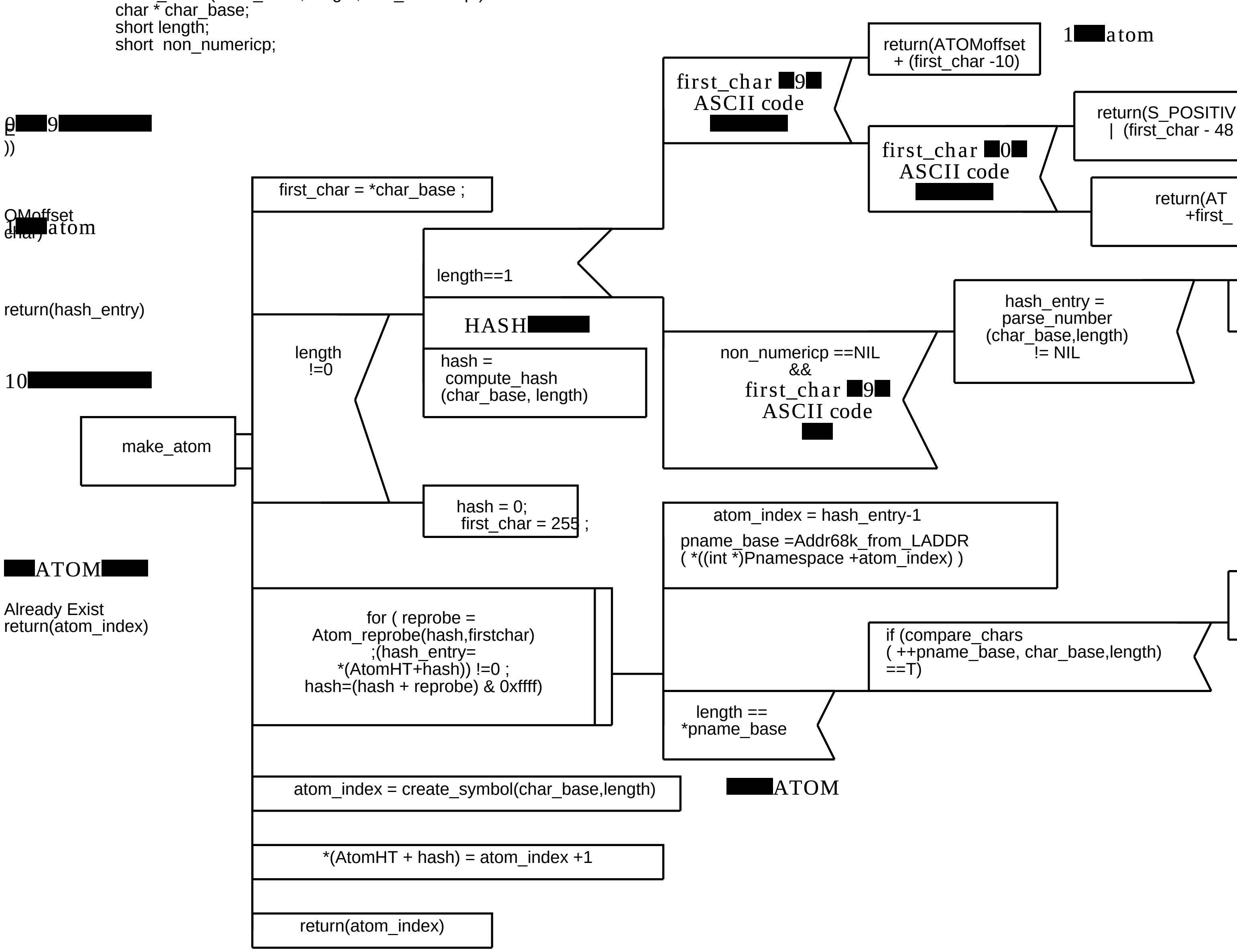
init_datatypes()



SPEC of LISTP(3Q)



```
make_atom ( char_base, length, non_numericp )
char * char_base;
short length;
short  non_numericp;
```



OP_cons ()

```
cons_x = GetLongWord(--CurrentStackPTR) ;
--CurrentStackPTR;
```
[redacted]

[redacted]Ref. cnt[redacted]

```
gclookup(ADDREF, TopOfStack);
gclookup(ADDREF, cons_x);
```

(TopOfStack == NIL_PTR)

CDRing NIL

((ListpDTD->dtd_nextpage != 0)
&&
(GetCONSCount(ListpDTD->dtd_nextpage) > 0))

ListpDTD[redacted]CONSPAGE [redacted]
1[redacted]Free[redacted]

[redacted]
[redacted]CONSPAGE[redacted]

```
new_page = ListpDTD->dtd_nextpage;
new_conspage =
 (struct conspage *)Addr68k_from_LPAGE(new_page);
new_cell = GetNewCell_68k(new_conspage);
new_conspage->count --;
new_conspage->next_cell = new_cell->cdr_code ;
```

```
new_cell->car_field = cons_x ;
new_cell->cdr_code = CDR_NIL ;
```
[redacted]

```
ListpDTD->dtd_cnt0++
```
[redacted]

[redacted]CONSPAGE[redacted]

```
new_conspage=next_conspage()
```

[redacted]CONSPAGE[redacted]

```
new_cell = GetNewCell_68k( new_conspage ) ;
new_conspage->count --;
new_conspage->next_cell = new_cell->cdr_code ;
```

```
new_cell->car_field = cons_x ;
new_cell->cdr_code = CDR_NIL ;
```
[redacted]

```
ListpDTD->dtd_oldcnt++;
```
[redacted]

TOS[redacted]CONSPAGE
[redacted]

```
new_page
 = POINTER_PAGE(TopOfStack);
```

(listp(TopOfStack)
&&
(GetCONSCount(new_page) > 0))

TOP[redacted]
1[redacted]Free[redacted]

On Page

TOS[redacted]CONSPAGE[redacted]

```
new_conspage=
 (struct conspage *)Addr68k_from_LPAGE(new_page);
new_cell = GetNewCell_68k( new_conspage ) ;
new_conspage->count --;
new_conspage->next_cell = new_cell->cdr_code ;
```

[redacted]

```
new_cell->car_field = cons_x ;
new_cell->cdr_code = CDR_ONPAGE | (LOLOC(TopOfStack)>>1);
```

```
ListpDTD->dtd_cnt0++
```
[redacted]

Diff. Page

[redacted]CONSPAGE[redacted]

```
new_conspage=next_conspage()
```

[redacted]CONSPAGE[redacted]2[redacted]

```
temp_cell = GetNewCell_68k( new_conspage ) ;
new_conspage->next_cell = temp_cell->cdr_code ;
new_cell = GetNewCell_68k( new_conspage ) ;
new_conspage->next_cell = new_cell->cdr_code ;
new_conspage->count -= 2;
```

CDR[redacted]

```
temp_cell->car_field = TopOfStack ;
temp_cell->cdr_code = 0 ;
```

CAR[redacted]

```
new_cell->car_field = cons_x ;
new_cell->cdr_code
= (LOLOC(LADDR_from_68k(temp_cell)) >> 1) ;
```

```
ListpDTD->dtd_oldcnt++;
```
[redacted]

TOS[redacted]

```
TopOfStack = LADDR_from_68k( new_cell )
```

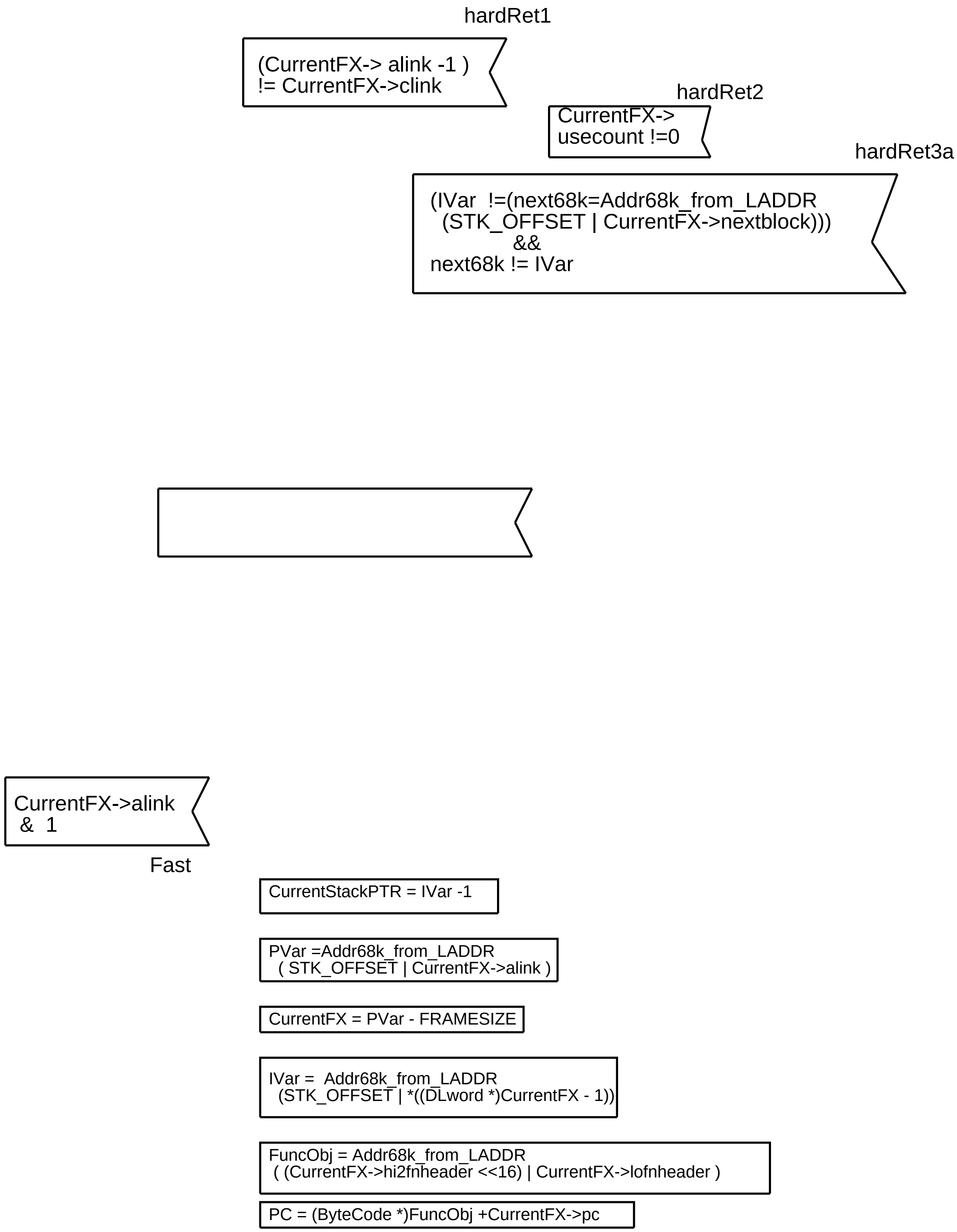
```
gclookup ( DELREF , TopOfStack )
```

```
PC++
```

Rev. 2-Apr-87

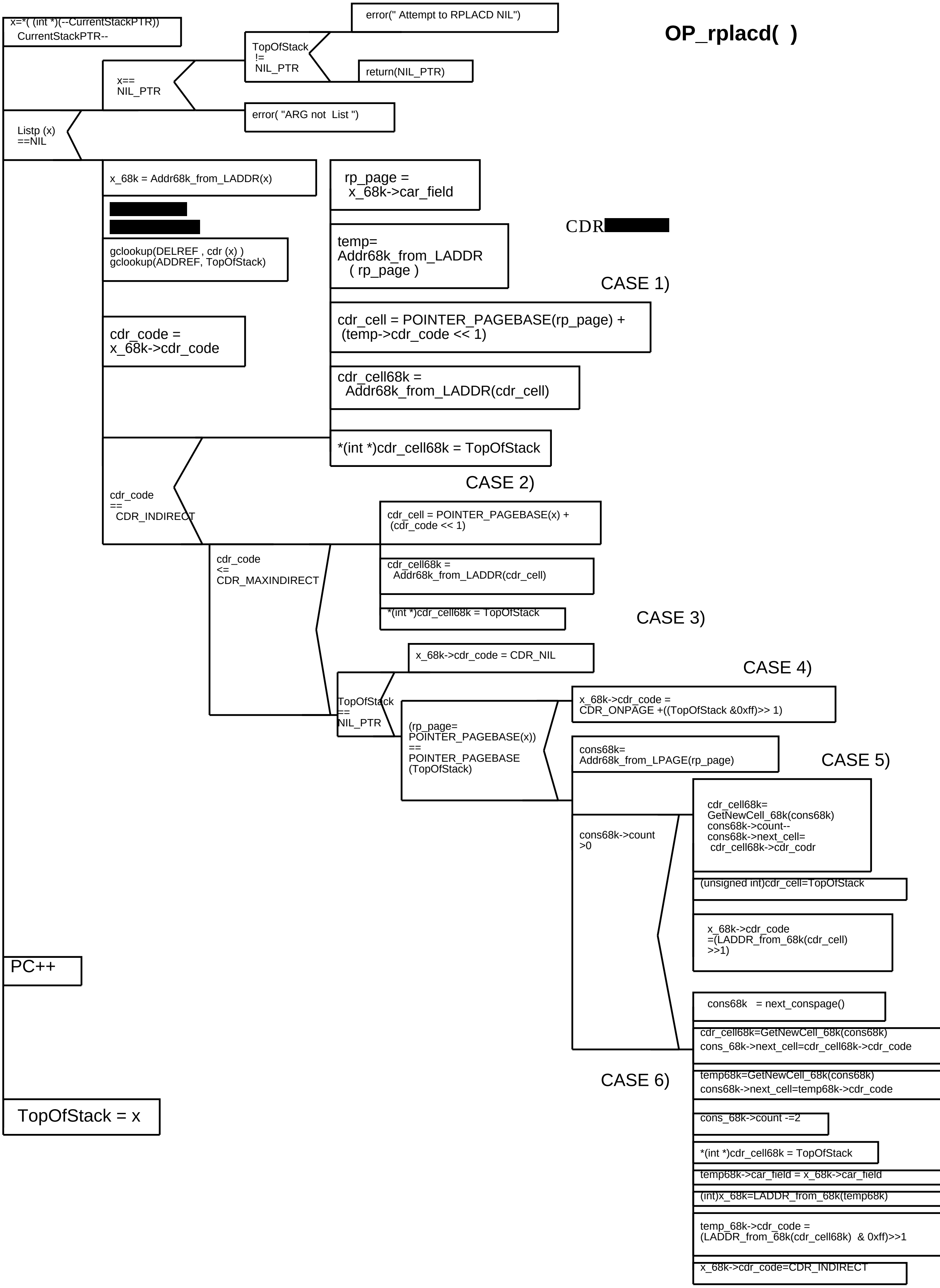
OP_fn

- CurrentFX->nextblock =(LADDR_from_68k(CurrentStackPTR) & 0x0ffff) -((*PC&0x0f)<<1) +3 — [redacted]IVar [redacted]
- IVar =Addr68k_from_LADDR(CurrentFX->nextblock) — Global IVar[redacted]
- PushCStack ; — TopOfStack[redacted]PUSH
- atom_index = (*(PC+1) <<8) | (*PC+2) — beta,gamma[redacted]ATOMINDEX[redacted]
- defcell68k =Get_DEFCELL68k(atom_index) — ATOMINDEX[redacted]DEFCELL[redacted]
- [redacted]
- (struct definition_cell *) defcell68k->ccodep==0
  - defcell68k = Get_DEFCELL68k(ATOM_INTERPRETER) — Interpreter[redacted]
  - PushStack(atom_index)
- FuncObj = Addr68k_from_LADDR((struct definition_cell *)defcell68k->defpointer) — [redacted]Compiled Object[redacted]
- (CurrentStackPTR+ FuncObj->stkmin)+2 > LastStackAddr
  - stackoverflow() — [redacted]
  - stackcheck()
    - stackoverflow()
- Spread Func?
- FuncObj ->na >=0
  - rest=((((*PC) & 0x0f) - FuncObj->na) — [redacted]FNHEAD [redacted]([redacted])
  - while (rest<0)
    - PushStack(NIL_PTR) rest++ — [redacted] NIL[redacted]PUSH [redacted]
  - CurrentStackPTR - (rest<<1)
- *(++CurrentStackPTR)=BF_MARK — Basic Fram [redacted]
- *(++CurrentStackPTR)=CurrentFX->nextblock
- *(++CurrentStackPTR)=FX_MARK — Frame Extension[redacted]
- CurrentFX=CurrentStackPTR
- CurrentFX->alink = LADDR_from_68k(PVar)
- CurrentFX->lofnheader = (defcell68k->defpointer)&0x0ffff
- CurrentFX->hi1fnheader = (defcell68k->defpointer)&0x0ff0000
- PVar=CurrentStackPTR=CurrentFX + FRAMESIZE — Global PVar[redacted]
- pv_num=FuncObj->pv
- while (pv_num>0) — PVar[redacted]
  - *((LispPTR *)CurrentStackPTR)=0xffff0000
  - CurrentStackPTR+=DLWORDSPER_CELL
  - *((LispPTR *)CurrentStackPTR)=0xffff0000
  - CurrentStackPTR+=DLWORDSPER_CELL
  - pv_num--



hardRet1
(CurrentFX-> alink -1)
!= CurrentFX->clink
hardRet2
CurrentFX->
usecount !=0
hardRet3a
(IVar !=(next68k=Addr68k_from_LADDR
(STK_OFFSET | CurrentFX->nextblock)))
&&
next68k != IVar
CurrentFX->alink
& 1
Fast
CurrentStackPTR = IVar -1
PVar =Addr68k_from_LADDR
(STK_OFFSET | CurrentFX->alink)
CurrentFX = PVar - FRAMESIZE
IVar = Addr68k_from_LADDR
(STK_OFFSET | *((DLword *)CurrentFX - 1))
FuncObj = Addr68k_from_LADDR
((CurrentFX->hi2fnheader <<16) | CurrentFX->lofnheader)
PC = (ByteCode *)FuncObj +CurrentFX->pc

OP_rplacd( )


x=*((int *)(--CurrentStackPTR))
CurrentStackPTR--
error(" Attempt to RPLACD NIL")
TopOfStack
!=
NIL_PTR
return(NIL_PTR)
x==
NIL_PTR
error("ARG not List ")
Listp (x)
==NIL
x_68k = Addr68k_from_LADDR(x)
[redacted]
gclookup(DELREF , cdr (x))
gclookup(ADDREF, TopOfStack)
cdr_code =
x_68k->cdr_code
rp_page =
x_68k->car_field
CDR [redacted]
temp=
Addr68k_from_LADDR
(rp_page)
CASE 1)
cdr_cell = POINTER_PAGEBASE(rp_page) +
(temp->cdr_code << 1)
cdr_cell68k =
Addr68k_from_LADDR(cdr_cell)
*(int *)cdr_cell68k = TopOfStack
CASE 2)
cdr_code
==
CDR_INDIRECT
cdr_cell = POINTER_PAGEBASE(x) +
(cdr_code << 1)
cdr_code
<=
CDR_MAXINDIRECT
cdr_cell68k =
Addr68k_from_LADDR(cdr_cell)
*(int *)cdr_cell68k = TopOfStack
CASE 3)
x_68k->cdr_code = CDR_NIL
CASE 4)
TopOfStack
==
NIL_PTR
x_68k->cdr_code =
CDR_ONPAGE +((TopOfStack &0xff)>> 1)
(rp_page=
POINTER_PAGEBASE(x))
==
POINTER_PAGEBASE
(TopOfStack)
cons68k=
Addr68k_from_LPAGE(rp_page)
CASE 5)
cdr_cell68k=
GetNewCell_68k(cons68k)
cons68k->count--
cons68k->next_cell=
cdr_cell68k->cdr_codr
cons68k->count
>0
(unsigned int)cdr_cell=TopOfStack
x_68k->cdr_code
=(LADDR_from_68k(cdr_cell)
>>1)
PC++
cons68k = next_conspage()
cdr_cell68k=GetNewCell_68k(cons68k)
cons_68k->next_cell=cdr_cell68k->cdr_code
CASE 6)
temp68k=GetNewCell_68k(cons68k)
cons68k->next_cell=temp68k->cdr_code
cons_68k->count -=2
*(int *)cdr_cell68k = TopOfStack
temp68k->car_field = x_68k->car_field
(int)x_68k=LADDR_from_68k(temp68k)
temp_68k->cdr_code =
(LADDR_from_68k(cdr_cell68k) & 0xff)>>1
x_68k->cdr_code=CDR_INDIRECT
TopOfStack = x

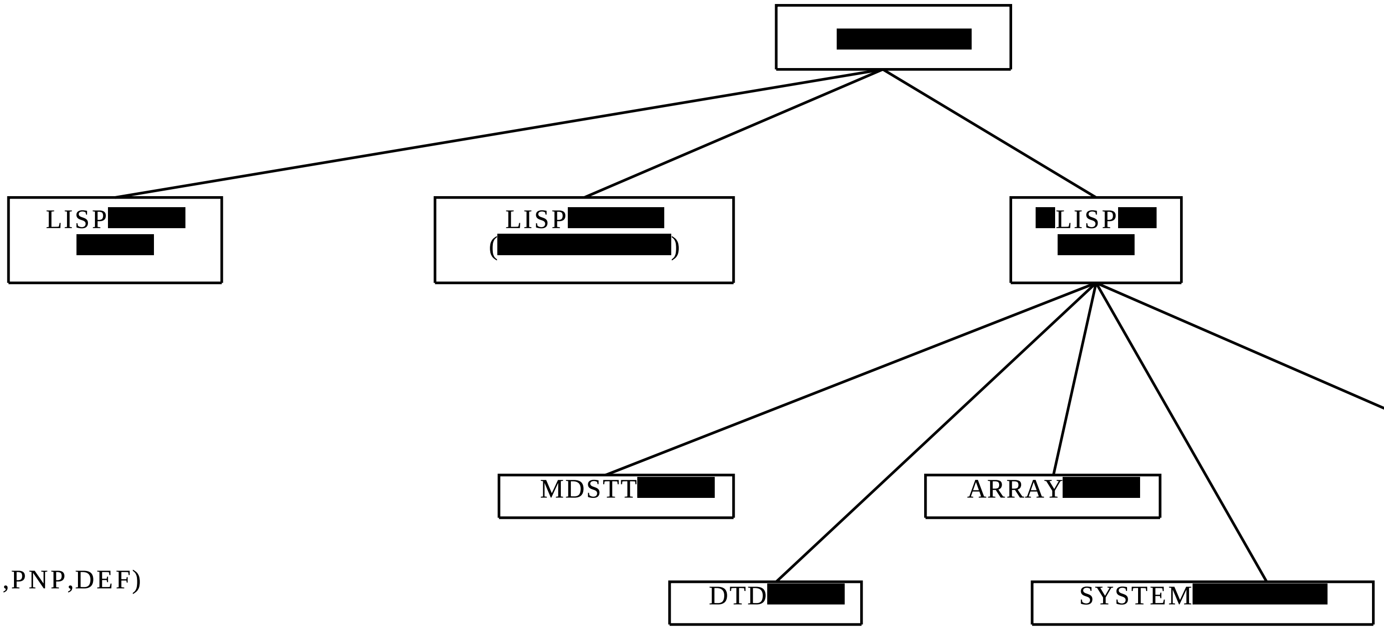
Revised 24 Mar. 198

7 T.Shimizu

6 1 ManWeek = 40 H

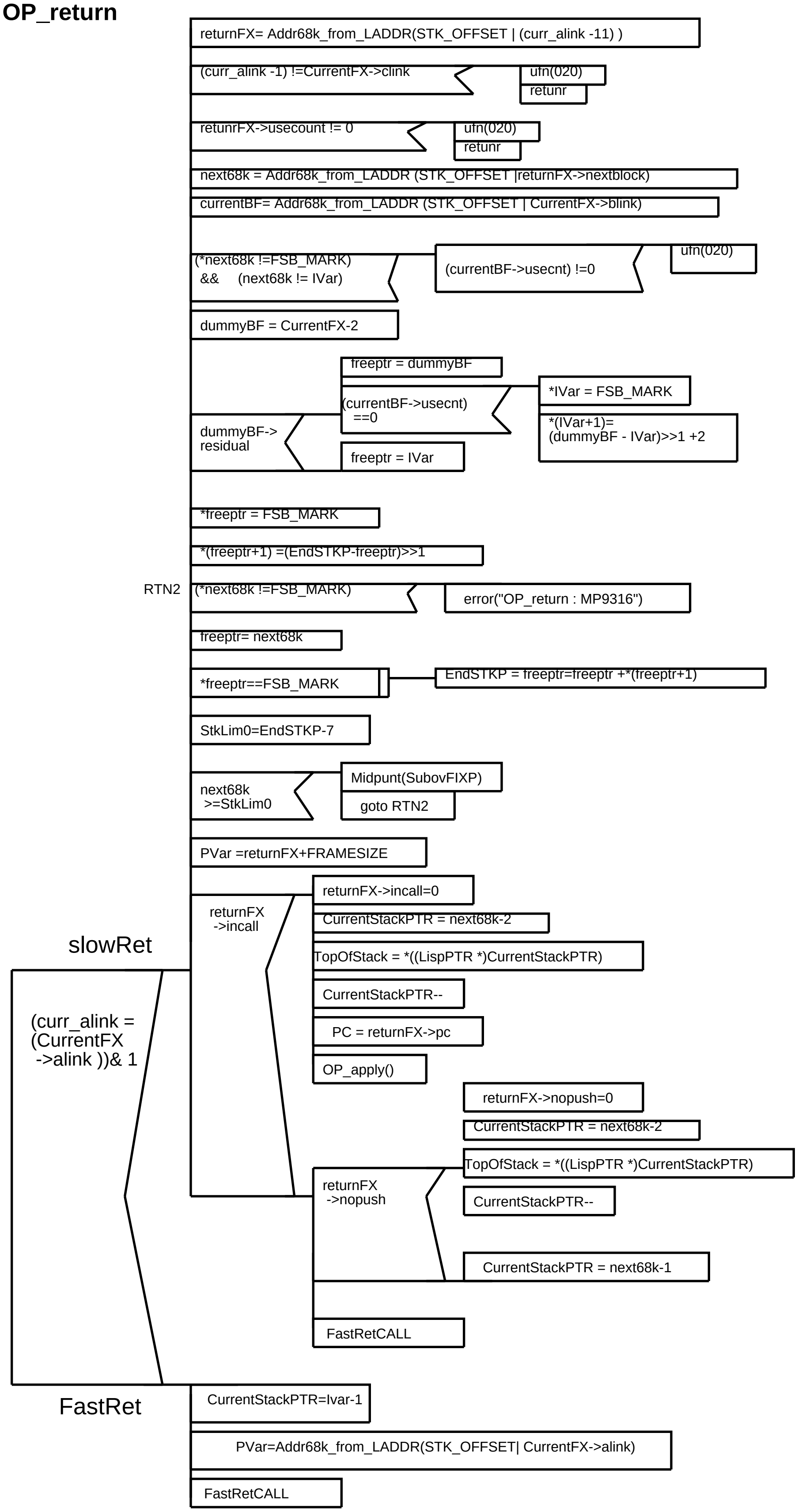
4

	Mar. 3	Mar. 4	Apr. 1	Apr. 2	Apr. 3	Apr. 4	May 1	May 2	May 3
[redacted]([redacted]1)	[redacted]		[redacted]						
[redacted]([redacted]2) ([redacted])					[redacted]				
[redacted] [redacted]([redacted]3)									
OPCODE[redacted] ([redacted] [redacted]1)	[redacted]([redacted])			[redacted](4/3)					
OPCODE[redacted] ([redacted] [redacted]2)	[redacted]([redacted])		1*Full 80 H	[redacted](4/3)					
[redacted](6/20 [redacted]) OPCODE[redacted] ([redacted])									
[redacted]	[redacted]	Coding	Debug						
[redacted] PROT([redacted]1)	[redacted]	Coding			[redacted]				
[redacted] PROT([redacted]2)		Coding			[redacted]	130H			[redacted]
[redacted] [redacted]						[redacted]			[redacted] & [redacted]([redacted]

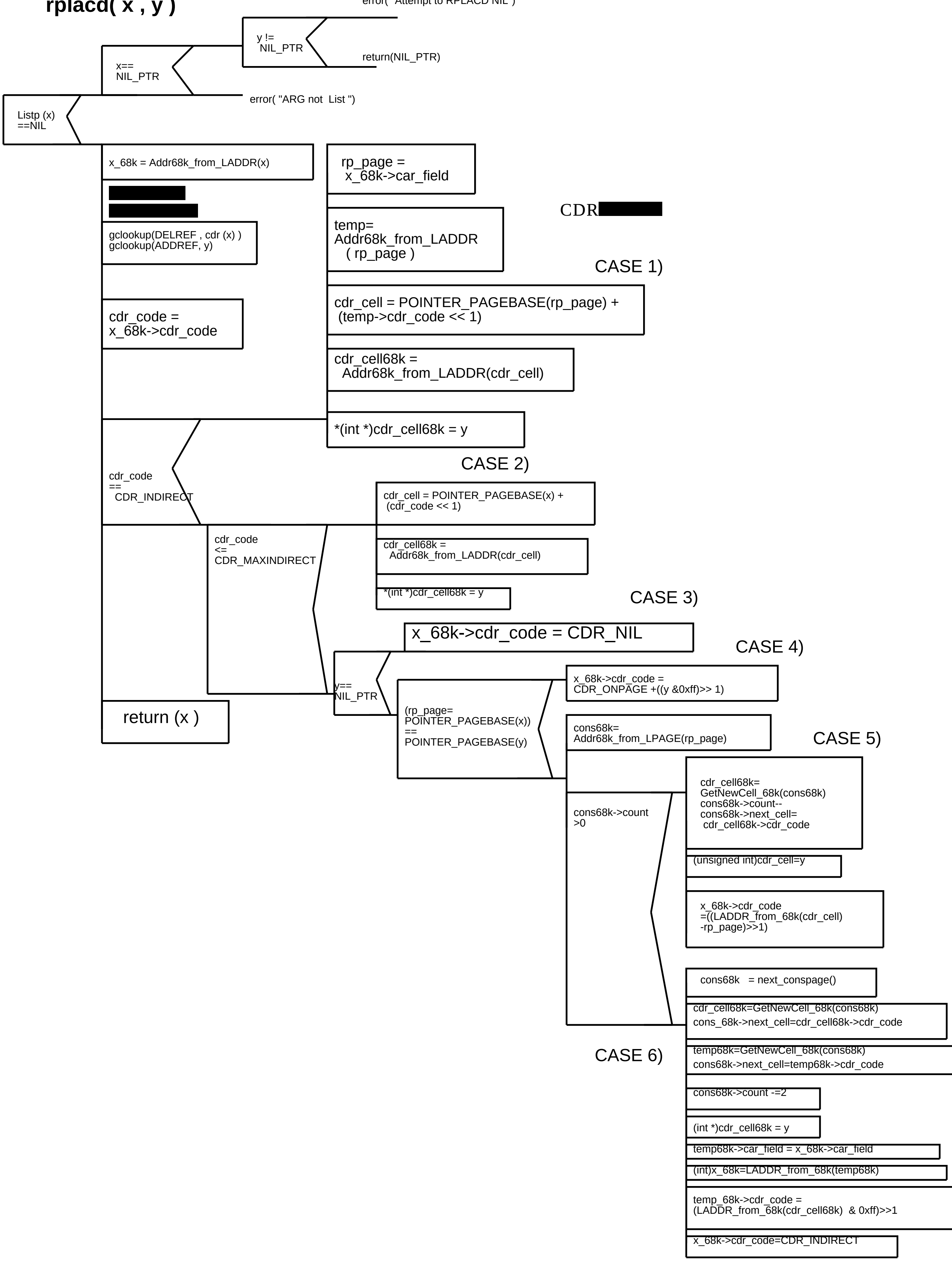


[redacted]
(PLIST,VAL,PNP,DEF)
[redacted]

OP_return


returnFX= Addr68k_from_LADDR(STK_OFFSET | (curr_alink -11))
(curr_alink -1) !=CurrentFX->clink
ufn(020)
retunr
retunrFX->usecount != 0
ufn(020)
retunr
next68k = Addr68k_from_LADDR (STK_OFFSET |returnFX->nextblock)
currentBF= Addr68k_from_LADDR (STK_OFFSET | CurrentFX->blink)
(*next68k !=FSB_MARK)
&& (next68k != IVar)
(currentBF->usecnt) !=0
ufn(020)
dummyBF = CurrentFX-2
freeptr = dummyBF
(currentBF->usecnt)
==0
*IVar = FSB_MARK
*(IVar+1)=
(dummyBF - IVar)>>1 +2
dummyBF->
residual
freeptr = IVar
*freeptr = FSB_MARK
*(freeptr+1) =(EndSTKP-freeptr)>>1
RTN2
(*next68k !=FSB_MARK)
error("OP_return : MP9316")
freeptr= next68k
*freeptr==FSB_MARK
EndSTKP = freeptr=freeptr +*(freeptr+1)
StkLim0=EndSTKP-7
next68k
>=StkLim0
Midpunt(SubovFIXP)
goto RTN2
PVar =returnFX+FRAMESIZE
returnFX->incall=0
returnFX
->incall
CurrentStackPTR = next68k-2
slowRet
TopOfStack = *((LispPTR *)CurrentStackPTR)
CurrentStackPTR--
(curr_alink =
(CurrentFX
->alink))& 1
PC = returnFX->pc
OP_apply()
returnFX->nopush=0
CurrentStackPTR = next68k-2
TopOfStack = *((LispPTR *)CurrentStackPTR)
returnFX
->nopush
CurrentStackPTR--
CurrentStackPTR = next68k-1
FastRetCALL
FastRet
CurrentStackPTR=Ivar-1
PVar=Addr68k_from_LADDR(STK_OFFSET| CurrentFX->alink)
FastRetCALL

rplacd(x , y)



error("Attempt to RPLACD NIL")
y !=
NIL_PTR
return(NIL_PTR)
x==
NIL_PTR
error("ARG not List ")
Listp (x)
==NIL
x_68k = Addr68k_from_LADDR(x)
[redacted]
gclookup(DELREF , cdr (x))
gclookup(ADDREF, y)
cdr_code =
x_68k->cdr_code
rp_page =
x_68k->car_field
temp=
Addr68k_from_LADDR
(rp_page)
CDR[redacted]
CASE 1)
cdr_cell = POINTER_PAGEBASE(rp_page) +
(temp->cdr_code << 1)
cdr_cell68k =
Addr68k_from_LADDR(cdr_cell)
*(int *)cdr_cell68k = y
CASE 2)
cdr_code
==
CDR_INDIRECT
cdr_cell = POINTER_PAGEBASE(x) +
(cdr_code << 1)
cdr_cell68k =
Addr68k_from_LADDR(cdr_cell)
*(int *)cdr_cell68k = y
cdr_code
<=
CDR_MAXINDIRECT
CASE 3)
x_68k->cdr_code = CDR_NIL
CASE 4)
y==
NIL_PTR
(rp_page=
POINTER_PAGEBASE(x))
==
POINTER_PAGEBASE(y)
x_68k->cdr_code =
CDR_ONPAGE +((y &0xff)>> 1)
cons68k=
Addr68k_from_LPAGE(rp_page)
CASE 5)
return (x)
cons68k->count
>0
cdr_cell68k=
GetNewCell_68k(cons68k)
cons68k->count--
cons68k->next_cell=
cdr_cell68k->cdr_code
(unsigned int)cdr_cell=y
x_68k->cdr_code
=((LADDR_from_68k(cdr_cell)
-rp_page)>>1)
cons68k = next_conspage()
cdr_cell68k=GetNewCell_68k(cons68k)
cons_68k->next_cell=cdr_cell68k->cdr_code
CASE 6)
temp68k=GetNewCell_68k(cons68k)
cons68k->next_cell=temp68k->cdr_code
cons68k->count -=2
(int *)cdr_cell68k = y
temp68k->car_field = x_68k->car_field
(int)x_68k=LADDR_from_68k(temp68k)
temp_68k->cdr_code =
(LADDR_from_68k(cdr_cell68k) & 0xff)>>1
x_68k->cdr_code=CDR_INDIRECT

\SMASHLINK(CALLER,ALINK,CLINK)

; Smashes caller’s ALINK and/or CLINK
with ALINK and CLINK

caller == NIL
caller=myakink()

oldalink=caller->alink
oldclink=caller-> clink

alink !=NIL
alink != clink
|| alink != oldclink
incusecount(alink)
caller->alink=alink

; Don’t increment twice
if ALINK comes out same as CLINK

clink!=NIL
alink ||
(clink != oldalink)
incusecount(alink)
caller->clink = clink
decusecount(oldclink)

; If we’re only setting the CLINK,
and we’re setting it to be the same as the ALINK,
don’t bump count

(oldalink !=
oldclink) && alink
decusecount(oldalink)

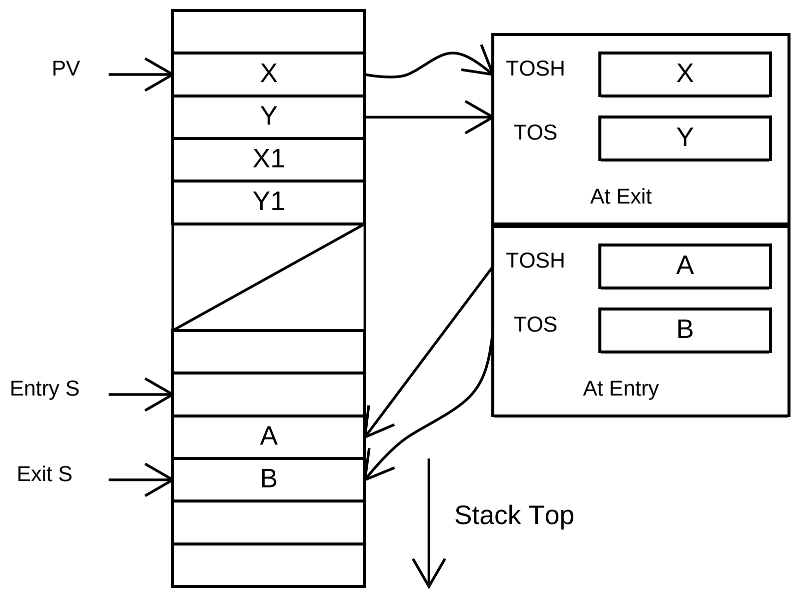
; must be careful to increment any
use counts before decrementing any

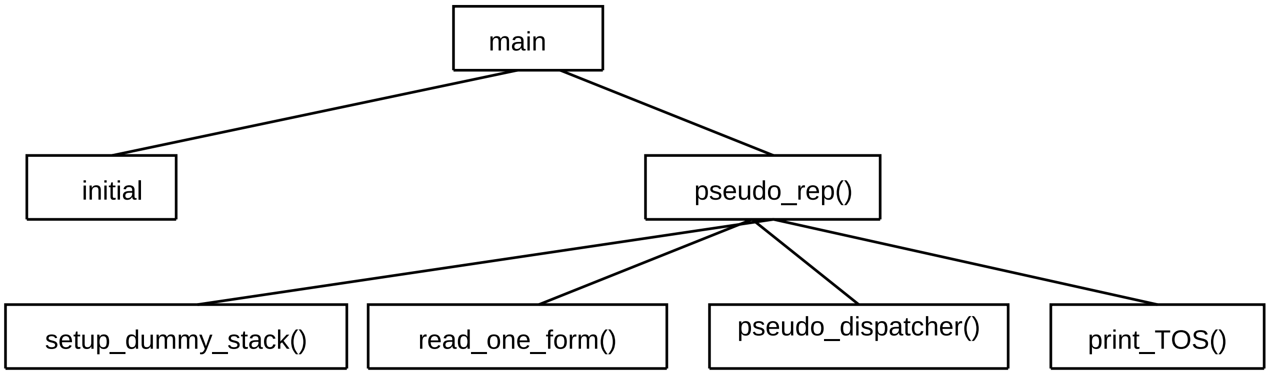
!alink
alink=oldalink

!clink
clink=oldclink

(alink==clink) &&
(clink->usecount==0) &&
((blink=caller->blink)==caller-2) &&
(blink->ivar ==clink->nextblock) &&
(blink->usecnt==0) &&
(!clink->nopush) &&
(!clink->incall)
caller->slowp=NIL

;; We have made CALLER fast again
: its alink and clink are same,
usecnt of blink and caller are normal,
bf is contiguous with CALLER and
CALLER’s caller


PV
Entry S
Exit S
X
Y
X1
Y1
A
B
TOSH
X
TOS
Y
At Exit
TOSH
A
TOS
B
At Entry
Stack Top


main
initial
pseudo_rep()
setup_dummy_stack()
read_one_form()
pseudo_dispatcher()
print_TOS()

SourceFiles.tedit

The following is the list of files in either the Carol sysout or the current lispcore sysout, showing those which we might not want to release the sources of. The notes "<removed>" and "<added>" indicate files that were removed or added to the lispcore loadup between the two versions.

Files in Lisp.sysout:

10MBDRIVER	
AARITH	
ABASIC	
ACODE	
ADDARITH	
ADIR	
ADISPLAY	
ADVISE	
AERROR	
AFONT	
AINTERRUPT	
AOFD	
APRINT	
APS	<removed>
APUTDQ	
ASSIST	
ASTACK	
ATBL	
ATERM	
BOOTSTRAP	
BREAK	
BRKDWN	
BSP	
BYTECOMPILER	
CHAT	
CLISP	
CLISPIFY	
CLEARINGHOUSE	<added> sensitive for now (bvm)
COMMENT	
COMPATIBILITY	<added>
COMPILE	
COREIO	
COURIER	sensitive protocols (Raim)
DEDIT	
DEXEC	
DFILE	
DISKDLION	
DLAP	
DLIONFS	
DMISC	
DPUPFTP	
DSPRINTDEF	
DTDECLARE	
DWIM	
DWIMIFY	
EDIT	
FILEIO	
FILEPKG	
FLOPPY	
FONT	
FPPATCH	<removed>
HELPDL	
HIST	

HLDISPLAY	
HPRINT	
IMAGEIO	
INSPECT	
INTERPRESS	
IOCHAR	
LEAF	
LLARITH	
LLARRAYELT	
LLBASIC	
LLBFS	
LLCHAR	
LLCODE	
LLDATATYPE	
LLDISPLAY	
LLETHER	
LLFAULT	
LLFCOMPILE	
LLFLOAT	
LLGC	
LLINTERP	
LLKEY	
LLNEW	
LLNS	
LLREAD	
LLSTK	
LLSUBRS	
LOADFNS	
MACHINEINDEPENDENT	
MACROAUX	
MACROS	
MASTERSCOPE	
MATCH	
MENU	
MISC	
MOD44IO	
MSANALYZE	
MSPARSE	
NEWPRINTDEF	
NSFILING	definitely proprietary (bvm)
NSPRINT	<added>
PASSWORDS	
PCALLSTATS	<removed>
PMAP	
POSTLOADUP	
PRESS	
PRETTY	
PROC	
PUP	
RECORD	
SPELL	
SPP	
TRSERVER	
TTYIN	
UNDO	
VOLUMEALLOCATIONMAP	
VOLUMEFILEMAP	
WBREAK	
WEDIT	
WINDOW	
WTFIX	

Additional files in Full.sysout:

```
AREDIT
ATTACHEDWINDOW
EDITBITMAP
FILEBROWSER
GRAPEVINE           potentially sensitive (bvm)
GRAPHER             <added>
HASH                <added>
ICONW
IMAGEOBJ
LAFITE              potentially sensitive (bvm)
LAFITEBROWSE        <added> potentially sensitive (bvm)
LAFITEMAIL          <added> potentially sensitive (bvm)
LAFITESEND          <added> potentially sensitive (bvm)
MAILCLIENT          potentially sensitive (bvm)
MTP                 potentially sensitive (bvm)
RDSYS
READNUMBER
READSYS
REMOTEVMEM
SAMEDIR             <added>
SINGLEFILEINDEX     <added>
SPY                 <added>
TEDIT
TEDITABBREV
TEDITCOMMAND
TEDITFILE
TEDITFIND
TEDITHCPY
TEDITHISTORY
TEDITLOOKS
TEDITMENU
TEDITSCREEN
TEDITSELECTION
TEDITWINDOW
TEXTOFD
TFBRAVO
VMEM
WHEREIS             <added>

Other files:

MAINTAIN            potentially sensitive (bvm)


===============

The following are the messages which lead to the creation of the above list:
```

Date: 11 Jun 84 15:42 PDT
From: Sannella.pa
Subject: Restricted Sources Files?
To: LispCore^.pa
cc: Sannella.pa
Reply-To: Sannella.pa

Beau has asked me to collect a list of the source files that we want to give to customers. Please send me (Sannella) the names of any source files you are responsible for which contain sensitive or proprietary information that should not be let out to the public at large. For each file, I would also like a few words of description of what it contains --- a sentence would be fine. I will compile the results, and send out another message. Thanks.

----- Next Message -----

Date: 12 Jun 84 10:58 PDT
From: vanMelle.pa

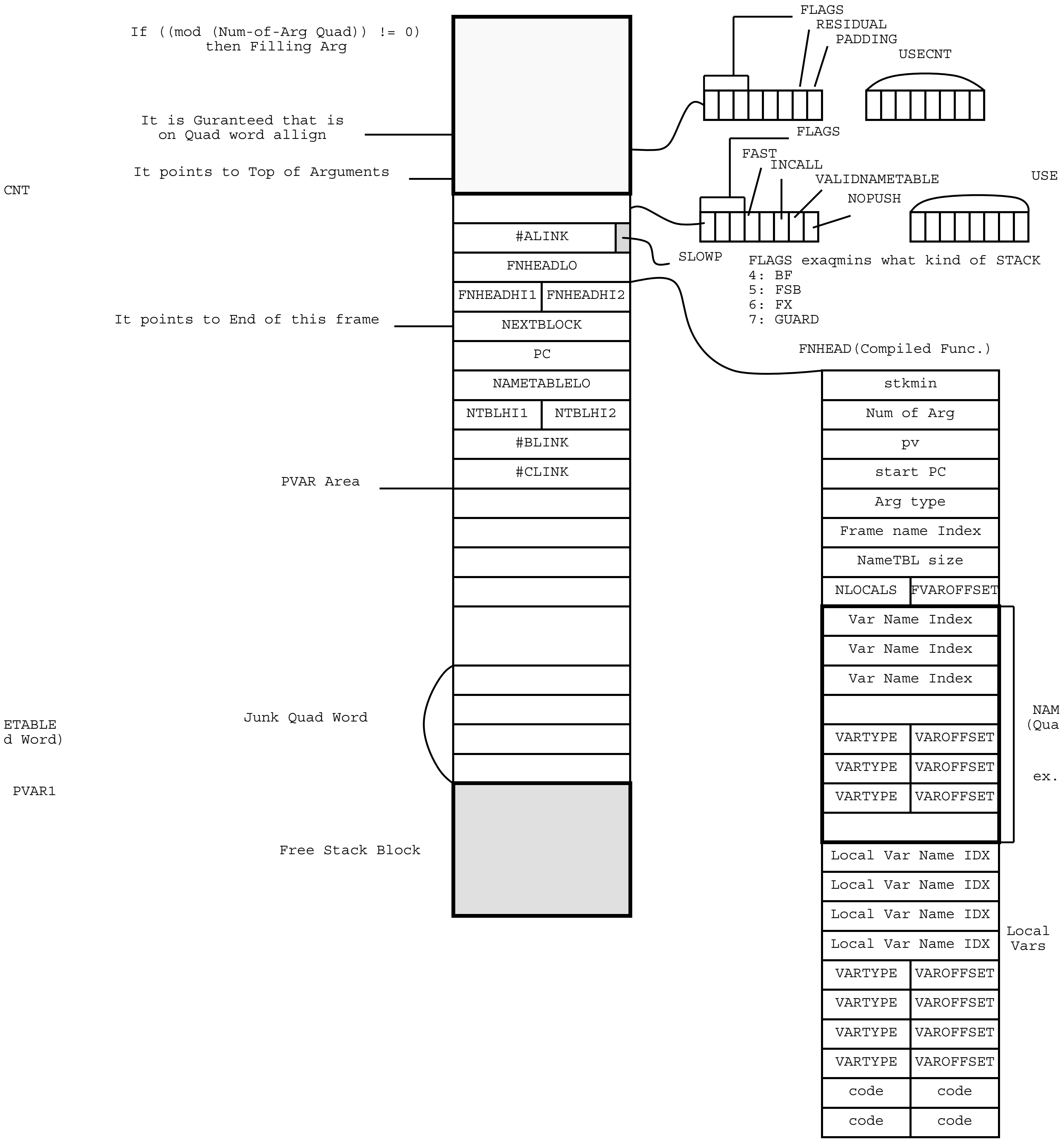
NSFILING is definitely proprietary (implementation of Filing protocol).

I would also consider CLEARINGHOUSE sensitive for now. Even though Clearinghouse protocol is now public, I think there may still be some things in that file that aren’t.

All the LAFITE* files, as well as GRAPEVINE, MAILCLIENT, MTP and MAINTAIN are potentially sensitive. Of course, no customers are even getting the dcom’s of those, so there would be no reason to give them sources.

Bill

Stack Contents (if Argments gives as 2) revised 6-Feb-87



Stack and Function Header Format

original: August 15, 1981 by Bill van Melle, Larry Masinter
reformatted for implementor's manual: July 2, 1985 by Ron Fischer
revised: August 1, 1985 by Bill van Melle
revised: July 16, 1988 by Frank Shih (minor note re: Maiko)

Stack structure

The stack segment in the Interlisp-D virtual memory consists of a series of stack *blocks*, which represent frames for active invocations of Lisp functions and free space left over. There are actually four specific kinds of blocks that can appear in the stack space: basic frames (abbreviated BF), frame extensions (abbreviated FX), free blocks (FSB), and guard blocks (GUARD). Each BF holds the arguments of a particular function call, while the FX holds other locally-bound variables and temporary storage for the function. A BF and FX are created at each call or entry, and released upon return or exit. A FX and its associated BF will simply be referred to as a *frame*.

Basic frame

A Basic Frame consists of the n arguments to the function being called, possibly followed by a cell of padding, followed by the BF word containing flags and a pointer to the first argument. The BF word is quadword-aligned. Every frame extension is immediately preceeded by a BF, either the actual BF, or by a "dummy" BF pointer in which the R bit is on and the IVAR pointer is valid. Only "slow" FX's can be preceded by a dummyBF.

```
IVAR:   *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |       0       |                  V a l u e 1                  |
        |      ...      |                      ...                      |
        |       0       |                  V a l u e n                  |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
BF:     |1 0 0 0 0 0|R|P|   Ext. cnt.   |             IVAR              |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

The fields are interpreted as follows:

IVAR — The 2nd word of the BF cell points back at the IVAR.

R — "Residual". This bit is on in "dummy" basic frames which are actually only 2-word quantities. Normal function call leaves the bit off. Dummy BF's are created when an FX has to move or be copied (on stack overflow or returning to a frame with non-zero use count). It's not clear these are really necessary—one could move/copy the entire frame (BF+FX).

P — "Padded". If 1, the actual number of arguments in the frame is 1 less than the size would indicate.

USECNT — Number of frame extensions (less 1) pointing at this BF. Initially 0.

Frame extension

A Frame extension consists of 10 words of control information, followed by storage for locally bound variables (PVARs), then binding slots for any variable referenced freely by the function, followed by dynamic storage ("the stack").

```
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

```
FX:     |1 1 0|F|-|C|V|N|   Use count   |          Alink=oldPVar       |X|
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |       fn header lo            |  undefined    | fn header hi  |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |      next (= TOP+2)           |         PC                    |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |       (name table lo )        |  undefined    |(nametable hi) |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |            (Blink)            |            (Clink)            |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
PVAR:   |1|          undefined        |1|    undefined                  |
        |1|             ...           |1|       ...                     |
        |1|          undefined        |1|    undefined                  |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
TEMP:   |       0       |              Temporary values                 |
        |      ...      |                     ...                       |
Top:    |       0       |              Last temp. value                 |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

The fields are interpreted as follows:

`F` "Fast". This frame binds no variables. This bit is (optionally) copied from the definition cell (see below).

`C` "InCall". Used by microcode to denote a frame interrupted in the middle of function call. Returning to a function with `C` on should invoke `APPLYFN` immediately instead of doing the return (the N bit is still valid).

`V` "NameTableValid". The "name table pointer" field is valid (see below).

`N` "NoPush". When returning to this frame, throw out the return value (used by punts, interrupts.)

The bits `C` and `N` need only be checked in a frame when performing a "slow return" to it. It is guaranteed that they will not be on in a frame which can be fast-returned to. The `V` bit is tested by free variable lookup.

`ALink` This field (with the low bit denoting "slow return") contains the `ALink` field of this frame, i.e., the pointer followed when chasing free variable references. The `ALink` points to the `PVAR` of the relevant frame, i.e., `FX`+10. The `ALink` field is also the `CLink` when the `X` (slow) bit is off. The `CLink` indicates the control chain, i.e., the frame to be returned to by this one.

`X` "SlowReturn". Must be set if the returnee is not contiguous with this this frame's `BF`, this frame's `BF` is not contiguous with its `FX`, or if there is a need to set the `CLink` independently of the `ALink`. Thus, if set, (a) the `BLink` and `CLink` fields of this `FX` are valid, and (b) when returning from this frame, the microcode for `RETURN` must do the "slow" case (usually punting, unless it's really a fast case after all).

`Fn Header` This is the pointer to the code base of this function (see Function Header format). Note that the high and low words of the function header are swapped.

`next` Points at the next stack word following this `FX`. Not valid while control is in this frame, of course—it is set when the frame is "closed out" by a function call or context switch of some sort.

`PC` Byte offset from FnHeader of next executable byte of this `FX`.

`NameTable` Where to find the table of names of arguments, prog variables, and free variable names. If the `V` bit in the flag word is off (normal case), this field is undefined, and the FnHeader field is used (the table of names is inside the function header); otherwise this field points at something that has the same format as a function header (see below).

BLink When X is set, points at the BF for this frame. When X is not set, BF=FX-2.

CLink When X is set, points at the returnee for this frame. When X is not set, CLink=ALink. CLink, like ALink, points at the PVAR area of the FX rather than at the beginning of the FX.

PVAR This area contains PROG/local variable values and free variable binding pointers. Initialized with all ones in the left half, which denotes {variable not bound} for PROG variables, and {variable not looked up} for free variables (see below). PVAR is quad-aligned (hence so are FX+1 and BF), and contains an even number of cells (so that TEMP is also quad-word aligned).

TEMP contains the temporaries (dynamic stack space) of the function. The first two cells of this region are garbage, for the benefit of the Dolphin hardware stack reloading. TEMP is quadword aligned, if necessary by padding out the PVAR area.

When a function is called, its arguments appear at the end of the caller's FX. The implementation is designed so that the bookkeeping for a BF appears at its high end, and for a FX at its low end, so that the arguments can be made into a new BF without having to copy them. After thus fabricating a new BF and shortening the old FX, the new FX is created. Upon return, the function value ("top of stack", which appears at the end of the returner's FX) is preserved while the returner's frame is deleted; then the value is pushed onto the caller's FX and execution resumes in the caller.

PVAR Slots and FVAR (Binding) Slots

When a variable is bound, via the BIND opcode, the corresponding slot in the PVAR area is filled in with the value:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|0|             |          value                                |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

When a free variable is found during free variable lookup, the variable's binding slot is filled in with:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|  binding pointer (even)     |0|  bind-addr-hi | bind-addr-hi  |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

This two word quantity is the (swapped) address of the location where the binding of the variable actually occurs. If the variable is not bound on the stack, this is the location of the variable's top level value, or possibly some other piece of storage. Note that all legitimate pointers are double-word aligned, and so have the low order bit turned off.

Note: on Maiko, the bind-addr-hi is *not* duplicated, i.e.:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|  binding pointer (even)     |0|       0       | bind-addr-hi  |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

The only way to tell between the PVAR and the FVAR region is to look in the name table (or function header if V is off): (PV+1)*2 is the number of cells (doublewords) in the PVAR region; NLOCALS is the number of those which are PVAR slots; the rest are FVAR slots or padding.

FVAR_ (the free variable assignment opcode), tests whether the binding address of the variable is in stack space. If so, it can do the assignment directly. If not, it must do the assignment using RPLPTR, i.e., decrement the reference count of the old value and increment the reference count of the new. This can be done by punting to the ufn if desired: the arguments to the ufn are the new value and the address (in normal order) of the binding location.

Note that this architecture does not specify anything about where non-stack bindings can be. Ordinarily, if the binding pointer is not in stack space, it is the top-level value cell of the variable, but it can also be used for closures, etc. to allow data structure manipulation to look like variable reference. The only current instance of this use is that the binding of RESETVARSLST at the top level of each process is actually a pointer into the process handle, for the benefit of cleaning up after a HARDRESET.

Free blocks

The other two stack block types are used to manage free space on the stack and as markers for book-keeping purposes. They are:

Ordinary Free block

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|1 0 1 0 0 0 0 0|       0       |              Size             |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

Guard block

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|1 1 1 0 0 0 0 0|       0       |              Size             |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

The only difference between Free blocks and Guard blocks is that microcode is guaranteed not to "use" Guard blocks. They are used at the end of stack regions. Size is in words.

Stack blocks may be discriminated by selecting the leftmost three bits of the flags into:

000b	Ordinary pointer (not a flag word)
100b	Basic frame
101b	Free block
110b	Frame extension
111b	Guard block

Function format

1. Function definition cells

Function definition cells are either:

Interpreted:
```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|     0         |           ptr to definition (list/NIL)        |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

Compiled:
```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|1|F|Aty|   0   |           ptr to definition (array block)     |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

Compiled definitions have 1 in the high order bit. The `Aty` field contains the ‘argtype’ of the function, and is used only by the `CHECKAPPLY*` opcode. The `F` bit is set if the corresponding `NTSIZE` of the definition is zero, i.e., the function has an empty name table. Undefined functions appear as an ’Interpreted’ function with definition = `NIL`.

2. Function header

A compiled function begins with a function header and a name table, which contains variable names; function names and other constants are stored in-line in the code. The first 8 words of the function are called the function header. The function header is quadword aligned. Format:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|               STK             |               NA              |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|               PV              |             START             |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|-|-|Aty|   0   |            function name                      |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|              NTSIZE           |    NLOCALS    |  FVAROFFSET   |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

`NA` The number of arguments this function expects. For `LAMBDA*` functions (arbitrary number of args, no adjustment) this field is ÿ1.

`PV` Number of quadwords required for the `PVAR` region, less 1, i.e., (#PVars + #Fvars + 1)/2 ÿ 1.

`STK` Amount of stack space required to call this function:

2*(max(NA, 0) + (PV+1) * 2) + FrameOverhead + MinExtraStackWords.

max(NA, 0) is maximum number of extra (doubleword) pushes to fill in unsupplied arguments; FrameOverhead=12 (includes BF overhead); MinExtraStackWords is the number of words microcode requires before punting, currently 32 (for D0 HStack microcode).

`START` Initial PC of the function.

`Aty` Argtype of the function: 0 = `LAMBDA`, 2 = `LAMBDA*`, 1 = `NLAMBDA`, 3 = `NLAMBDA*`). ignored by microcode, but transcribed to the definition cell, where `CHECKAPPLY*` tests it.

`NTSIZE` Name table size (in words). This is a multiple of 4.

`NLOCALS` Number of `PROG` variables.

`FVAROFFSET` Offset (from start of header) of first `FVAR` in name table.

This is then followed by the specvar Name Table (not to be confused with the `NAMETABLE` field of the `FX`, which always points at something looking like a function header), which is actually two parallel tables.

The first table contains atom numbers ("value index"), terminated with enough zeros to fill out a quadword, but always at least one word of zero. Thus the size of the table in words is `NTSIZE`+1 rounded up to a multiple of 4. Note that in the special case where `NTSIZE` is 0, there is still a quad-word of zeros. The name table is arranged with `PVAR` names in reverse order of binding, `IVAR` names, and then `FVAR` names. Thus, when free variable lookup scans the name table in order, it will find the most recent bindings first.

The second table (which starts `NTSIZE` words beyond) contains entries of the form:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|vty|     0     |     offset    |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

vty codes are as follows:

00	This name corresponds to an `IVAR`.
10	This name corresponds to a `PVAR`.
11	This name corresponds to a `FVAR`.

Offset is a zero-based doubleword offset from the start of the corresponding section of the `BF` or `FX`. Both `PVAR` and `FVAR` offsets are relative to `PVAR`, i.e., `PVAR` *n* is followed by `FVAR` *n*+1.

Free variable lookup scans the first table for a match of the "looked up variable". If a match is found, the word at the same offset in the second table is fetched to determine whether the value is in the `IVAR` section (vty=0), in the `PVAR` section (vty=2) or pointed to by the `FVAR` section (vty=3).

The regular name table is followed by the "localvar argument" name table, which is not visible to the microcode. It is in the same format as the regular name table, but lists names of arguments to the function that are declared `LOCALVARS` and hence would otherwise be invisible. This table is for the benefit of `ARGLIST` and the debugger.

Streams and File Devices

Edited 30 Nov 84, van Melle
Edited 19 June 86, Jellinek
Edited 22 September 86, Jellinek
Edited 3 October 86, Jellinek

An Interlisp Stream is an object of datatype STREAM that is capable of performing, at the least, sequential input and/or output of bytes. Some streams can do much more. Streams are used for access to open files, for writing to the display, for chatting to remote hosts, and whatever other uses people come up with. This document describes how one goes about defining a new device, the meanings of the record fields of the STREAM and FDEV datatypes, and anything else that seemed relevant at the time.

The implementation of Streams is strongly object-oriented. Every STREAM has a pointer to a device (the datatype FDEV), which contains a vector of functions to be called when certain operations are required of the stream. There can be many streams with the same device. In the object-oriented terms of LOOPS, one can think of the device as a *class*, which provides a set of *methods* that implement class operations, and the streams as *instances*. Devices and streams also have local state, which might be thought of as class and instance variables. Declarations for STREAM and FDEV can be obtained by loading EXPORTS.ALL.

OPENSTREAM, CLOSEF, FORCEOUTPUT, READP, EOFP, GETEOFPTR, GETFILEINFO, SETFILEINFO, DIRECTORY, COPYBYTES, DELFILE, RENAMEFILE, FULLNAME are some of the Lisp functions called by the programmer that ultimately turn into operations at the device level. The descriptions that follow sometimes allude to these functions, and knowledge of how they operate may occasionally give the reader additional clues as to how the device operations work.

Typically, some part of the operation is handled by the ‘‘generic’’ file system code, which then calls on the device to handle that part of the operation that is device-specific. For example, the function OPENSTREAM takes the name of the file it is to open and fills in host and directory defaults, and decides which device handles such a file. It then calls on the particular device to actually open the file. After the file is opened, the generic file system code registers the file on (OPENP). As another example, operations involving open streams first coerce non-streams (e.g. filenames) to open streams before calling the device-specific operation. With the advent of multiple streams per file, the last two examples will soon no longer obtain.

Devices

A device is an object of type FDEV (so named for historical reasons: ‘‘File Device’’). The standard way to define a new device is to create such an object, by performing (**create** FDEV --), and then pass the newly created FDEV to the function \DEFINEDEVICE. \DEFINEDEVICE is the way a device ‘‘announces’’ itself to the generic file system.

(\DEFINEDEVICE *NAME DEV*) [Function]

Installs device *DEV*, giving it the name *NAME*. *NAME* must be an uppercase litatom. The generic file system code makes use of the name to locate the device that is willing to deal with files whose full name begins {*NAME*}. It is permissible to have more than one name map to the same device; this effectively provides device synonyms. Devices are encouraged, however, to always create file names using the canonical device name, independent of what name was passed in.

NAME can be NIL, in which case the generic file system code never consults the device directly. However, its EVENTFN is still run around Lisp exits, and it can be used as the device for a stream created by nonstandard methods.

If a device never wants to be invoked by name, and has no interesting EVENTFN or HOSTNAMEP methods, then there is no need to ever register it with \DEFINEDEVICE.

(\REMOVEDEVICE *DEV*) [Function]

Removes device *DEV* from the list of known devices, as well as any name that maps to that device.

(\GETDEVICEFROMNAME *NAME NOERROR DONTCREATE*) [Function]

Returns the device associated with *NAME*. *NAME* can be a litatom or string; it is coerced to uppercase, and if it begins with an open brace, is assumed to be a file name, from which the host name is extracted. If no such device is known, attempts to find one by polling the HOSTNAMEP methods of all known devices (see below); if a device is still not found, causes a FILE NOT FOUND error unless *NOERROR* is true. If *DONTCREATE* is true, it never attempts to create a device, just returns an existing device if there is one, NIL otherwise.

The fields of an FDEV are divided up into informational fields and ‘‘methods’’.

DEVICENAME — A pointer field, the name of the device, standardly a litatom. Use of this field is largely up to the device, but it is usually selected to be the name that appears inside braces in filenames opened on this device. For devices that do not support the notion of named files, DEVICENAME can be anything that the implementor cares to use for debugging assistance.

RESETABLE — A flag, true if (SETFILEPTR stream 0) can be performed. Currently unused.

RANDOMACCESSP — True if the stream is randomly accessable, i.e., if SETFILEPTR works on this kind of stream.

NODIRECTORIES — True if files opened on this device do not (usually) have a directory as part of their name. The principal use for this is by the CONN command, which will not try to connect to the user's home directory if given a host only, e.g., CONN {DSK}.

BUFFERED — True if streams of this sort are buffered in a manner compatible with the microcoded versions of BIN and BOUT. More specifically, BUFFERED implies that the device implements the GETNEXTBUFFER method. See description of buffered streams.

PAGEMAPPED — True if this stream is implemented by the pagemapped functions. All pagemapped streams are also buffered, so if this flag is true, so should be BUFFERED. See description of pagemapped streams.

FDBINABLE — True if streams on this device obey the rules for microcoded BIN whenever such stream is open for input access.

FDBOUTABLE — True if streams on this device obey the rules for microcoded BOUT whenever such stream is open for output access. Currently unused, as the spec needs revision.

FDEXTENDABLE — Special kind of FDBOUTABLE. Currently unused, as the spec needs revision.

DEVICEINFO — A pointer to arbitrary device-specific information. The standard use for this is to hold local state specific to one of several similar devices that share methods. For example, the Dolphin disk provides a separate FDEV for each partition of the machine; the DEVICEINFO field of each has pointers to the partition's directory and other information specific to files on that partition only.

OPENFILELST — A list of all streams open on this device. Not required; it's provided for convenience only.

The following fields are all pointer fields, and contain functions for implementing various device operations. Not all devices need have all fields filled in; the required ones are listed first and so indicated. Some ''required'' fields have defaults specified in the FDEV (or STREAM) record declaration, so the implementor need not explicitly fill those fields if the default is reasonable. Each field is presented with its arguments, in the style of a function definition; of course, it is the contents of the field, not the field name, that is the function. Using object-oriented terminology, the occupants of these fields are referred to as ''methods''. For example, ''the BIN method'' means ''the function that occupies the BIN field''.

One of the arguments to each method is usually either the device itself, or a stream open on the device, so that the device (and hence its DEVICEINFO) is usually accessible to all these functions. Arguments that are file names or patterns or pieces of file names can be either litatoms or strings, and already have their host and/or directory parts appropriately filled in from the connected directory defaults. The device may assume that the host field of the file name is indeed a name that the device has said it implements (see HOSTNAMEP). ''Full'' file names returned by these functions (or stored in the FULLFILENAME field of a stream) should be litatoms, and at least in the current implementation should be all uppercase.

Fields required of every device:

(HOSTNAMEP *HOSTNAME DEVICE*)

Called by the generic file system code when presented with a host name for which there is as yet no device defined. The function should return non-NIL if it ''recognizes'' *HOSTNAME*. There are two ways in which this method is invoked:

(1) To obtain a device for *HOSTNAME*, for example, so that a file can be opened on it. In this case, *DEVICE* is an already defined device (the one whose HOSTNAMEP method is being called), and the function should return either a new device, or T, meaning it is willing to take responsibility for this host name as well as any previous name under which the device was registered. In either case, the caller will install the returned device, or *DEVICE* if value was T, as the device to which *HOSTNAME* maps.

(2) As a pure predicate. In this case, *DEVICE* is NIL, and the function need only return T or NIL, indicating whether it believes that *HOSTNAME* is the name of a host.

In practice, the HOSTNAMEP method need only take care of the first case, since that also takes care of the second case. The second case is provided so that the device need not be created until there is an actual use for it, should the device wish to avoid unnecessary work. In practice it is rare that anyone tests a host name without subsequently needing to have the device created in full.

There are basically three kinds of devices in the system as distinguished by their HOSTNAMEP methods.

(1) Predefined devices with exactly one name, or strictly internal devices with no notion of name. For example, the CORE device always exists, and has exactly one name; the SPP device (a network byte stream) has no name (it supports no files directly). Such devices have a HOSTNAMEP method of NILL—the only name they ever go by is the one they gave to \DEFINEDEVICE, if any. This is the default.

(2) Devices that don't know ahead of time what their name will be, but for which there might be many incarnations. This is the model for remote file servers. The standard way of handling this case is to define a dummy device that has only a `HOSTNAMEP` method, and no name. When the `HOSTNAMEP` method gets called with a name that the device knows it can service, it creates a device by that name. If given a name that is a synonym of another name, it might just return the existing device of the canonical name (using `\GETDEVICEFROMNAME` to find the right device). In either case, the `HOSTNAMEP` method of the new device is usually `NILL`—the original device is the only one that worries about creating new instances of this class of device.

(3) Like (2), but all the different names are handled by a single device, which takes care internally of the multiplexing among, say, different remote hosts. `HOSTNAMEP` returns `T` in this case. This is usually clumsier than (3), so discouraged.

(`EVENTFN` *DEVICE EVENT*)

Called around Lisp exits, to allow the device to do any necessary cleaning up, clearing of caches, disconnects with remote hosts, revalidation of files, etc. *EVENT* is one of the following litatoms: `BEFORELOGOUT`, `BEFORESYSOUT`, `BEFOREMAKESYS`, `BEFORESAVEVM`, `AFTERLOGOUT`, `AFTERSYSOUT`, `AFTERMAKESYS`, `AFTERSAVEVM`, `AFTERDOSYSOUT`, `AFTERDOMAKESYS`, `AFTERDOSAVEVM`. The `AFTER`xxx events are all run when Lisp is booted from a memory image that resulted from a `LOGOUT`, `SYSOUT`, etc. The `AFTERDO`xxx events run when continuing Lisp in the same incarnation following the `SYSOUT`, etc. (there is no such event for `LOGOUT`, of course). The ‘‘after’’ events are called in the same order in which the devices were defined; the ‘‘before’’ events in the reverse order.

For example, the `BEFORELOGOUT` event for the Leaf remote file server devices performs a `FORCEOUTPUT` on all its open files and then breaks the connection with the file server. The `AFTER`xxx events for the Leaf devices calls `\REMOVEDEVICE` on itself to flush any connection between the name and the server (since names and addresses can change over exit), and then revalidates all of the device's open files. The `AFTER`xxx events for the Dorado disk device rebuilds its cache of the disk's directory.

There are a few devices in the system that exist only for their `EVENTFN`. In most cases, a simpler way to tell the system you want something performed around exit is to add your event function to the list `AROUNDEXITFNS` instead of going to the expense of defining a device for it. There is yet another list, `\SYSTEMCACHEVARS`, for handling a more specialized ‘‘around exit’’ operation: every time Lisp is booted, each of the variables in the list `\SYSTEMCACHEVARS` is set to `NIL`.

The following are required of all named devices, that is, devices that map from some hostname to the device, upon which files might be opened or otherwise manipulated:

(`DIRECTORYNAMEP` *HOST/DIR DEVICE*)

True if *HOST/DIR* is a valid directory name on *DEVICE*. Function should ideally perform recognition as well, and return the ‘‘true’’ name. For example, given ‘‘`{PHYLEX:}<LISP>`’’ as argument, it might return `{Phylex:PARC:Xerox}<Lisp>`. *HOST/DIR* might include a subdirectory name. The device should attempt to tell the truth about whether the subdirectory exists or not, though this may not be possible for devices with fake subdirectories. Defaults to `NILL`, i.e., device supports no directories. Used by the command `CONN` and the function `DIRECTORYNAMEP`.

(`OPENFILE` *NAME ACCESS RECOG PARAMETERS DEVICE*)

Used to implement the `OPENFILE` and `OPENSTREAM` functions. Opens the file named *NAME* on this device for access *ACCESS*, returning a `STREAM`. The stream is usually on *DEVICE* (its `DEVICE` field is *DEVICE*), but is not required to be. The arguments *ACCESS*, *RECOG*, *PARAMETERS* are as with the

`OPENFILE` function in the manual. Thus, if *NAME* does not include a version number, recognition is according to *RECOG*, which should be appropriately defaulted per *ACCESS* (`INPUT` implies `OLD`, `OUTPUT` implies `NEW`, `BOTH` implies `OLD/NEW`).

The argument *NAME* can also be a `STREAM`, which must be a closed stream. `OPENFILE` should ‘‘reopen’’ the stream. The value returned in this case may be a new stream (with the same name as the old), or the old stream (*NAME*) itself. It is likely that the specification will be changed at some point to require that the old stream itself be returned, suitably reopened.

The argument *PARAMETERS* is a list of pairs (*OPTION VALUE*). The most interesting *OPTION*s are as follows:

`TYPE` For new files, the type of the file (`TEXT` or `BINARY`). If this parameter is not specified, the value of the global variable `DEFAULTFILETYPE` (initially `TEXT`) should be used.

`CREATIONDATE` For new files, the date of its creation. The device should use this if at all possible instead of letting the creation date default to the current date and time.

`LENGTH` The intended length of the file, in bytes. This need not be accurate—it is only a hint that may allow smarter allocation. For example, if the device knows that it does not have room for a file of the specified length, it should immediately cause a `FILE SYSTEM RESOURCES EXCEEDED` error for the intended file.

`DON'T.CHANGE.DATE` For old files being opened for access `BOTH`, don’t change the creation date of the file. *ACCESS* = `BOTH` would normally imply that the content of the file is to change, and thus its creation date should be updated. Use of this parameter is a form of ‘‘cheating’’ to make it look as though the file had not changed. For example, the code that rewrites filemaps uses this parameter, since rewriting the filemap does not logically change the file’s content.

`SEQUENTIAL` If `T`, is a hint that the file will, or need, only be accessed sequentially, which may allow the device to open the file in a more efficient mode.

Any parameters that the device does not understand should be ignored, rather than be cause for an error. All devices are encouraged to support at least `TYPE` and `CREATIONDATE`.

The additional options `ENDOFSTREAMOP` and `BUFFERS` are handled by the generic file system code; specifying them is equivalent to calling `SETFILEINFO` (q.v.) immediately after the open.

Fine point about *ACCESS* = `OUTPUT`: this operation always produces a new, empty file, independent of whether its name is exactly the name of an existing file. That is, it replaces any old file by the same name. On opening, such a file has an end of file of zero. Of course, since *RECOG* defaults to `NEW` in this case, the name can only clash with an old file name if a version was explicitly specified, or *RECOG* is `OLD` or `OLD/NEW`. To open an old file for output but preserve its contents, i.e., only write over part of the file, one should open for *ACCESS* = `BOTH` (since to preserve the old contents one implicitly reads them).

Exception handling: If the desired file is not found, the `OPENFILE` method should return `NIL` rather than cause a `FILE NOT FOUND` error. This is so that the generic file system code can cause the error using the original file name, not the one packed with host and directory passed in to the `OPENFILE` method. The device should feel free to signal any other errors itself on failing to open the file, e.g., `FILE WON'T OPEN` for a busy file, `PROTECTION ERROR`, or `FILE SYSTEM RESOURCES EXCEEDED`. Ideally, this error should be signaled in a way that is resumable, i.e., so that a user could, in the break, take some action to remedy the condition and then type `OK` to continue. In most cases it suffices that all the internal functions below the `OPENFILE` be named with backslashes, so that the error code will choose to resume by reverting to the `OPENFILE` and trying again.

The device must keep track of the set of files open on it, for use in access-conflict detection, and as an assist to the user in closing "dropped" streams. This is done by the generic file system, employing the `REGISTERFILE` method, below.

(`CLOSEFILE` *STREAM*)

Closes *STREAM*. Performs all necessary ''finalization'' on *STREAM*, including doing a `FORCEOUTPUT` or equivalent if *STREAM* was open for output. De-registers *STREAM* from its device's list of open streams.

(`REOPENFILE` *NAME ACCESS RECOG PARAMETERS DEVICE OLDSTREAM*)

This is exactly like `OPENFILE`, except that it is called after `LOGOUT` (or other ''after'' events) by the device's `EVENTFN` on the name of any stream that was left open over exit. The idea is to maintain the illusion that the file really was open over `LOGOUT`, but check and make sure nothing changed. The generic file system code uses the `VALIDATION` field to test whether the file changed behind your back.

OLDSTREAM is the stream that was open before exit, and is supplied for the benefit of devices where there is no possibility that the file changed (e.g., {`CORE`}), so that they can just return *OLDSTREAM* directly. *OLDSTREAM* is also of use for those devices that have to cheat in order to maintain the illusion.

(`GETFILENAME` *NAME RECOG DEVICE*)

Performs ''recognition'' on *NAME*. That is, it returns the full name of the file that would be opened by `OPENFILE` in the indicated recognition mode, or `NIL` if the file is not found. It is not necessary that `OPENFILE` actually be capable of opening the file (there is no need to check protection, for example). Used by `INFILEP`, `OUTFILEP`, `FULLNAME`.

(`DELETEFILE` *NAME DEVICE*)

Deletes the file named *NAME*, returning its full name on success, `NIL` on failure. Recognition mode is implicitly `OLDEST`. Local devices, after recognizing the file, should make sure that it is not among the device's open files (open files cannot be deleted). This, OPENFILE, and RENAMEFILE are usually the only device methods that need to know anything about what files are open.

(`GENERATEFILES` *DEVICE PATTERN DESIREDPROPS OPTIONS*)

Enumerates files matching *PATTERN*. Returns a ''file generator object'' of the form (*NEXTFILEFN INFOFN* . *ArbitraryState*). This is described in more gory detail under **Directory Enumeration**.

(`RENAMEFILE` *OLD-DEVICE OLD-NAME NEW-DEVICE NEW-NAME*)

(A method of *OLD-DEVICE*.) Renames the file named *OLDNAME* on device *OLD-DEVICE* to have name *NEWNAME on NEW-DEVICE*. Returns the full name of the new file if successful, `NIL` if not. Recognition mode is implicitly `OLD` for *OLDNAME*, `NEW` for *NEWNAME*. The generic file system code always invokes this method to implement the function `RENAMEFILE`. If *OLD-DEVICE* and *NEW-DEVICE* are not `EQ`, the `RENAMEFILE` method can call `\GENERIC.RENAMEFILE` to do the job; `\GENERIC.RENAMEFILE` is defined to copy *OLDNAME* to *NEWNAME* and then delete *OLDNAME*. Defaults to `\GENERIC.RENAMEFILE`. RENAMEFILE needs to check if OLD-NAME is open; only closed files can be renamed.

(`OPENP` *FILENAME ACCESS DEVICE*)

Returns all of *DEVICE*'s open streams with name *FILENAME* that are open in mode *ACCESS*. *FILENAME* and/or *ACCESS* may be `NIL`, which matches all names and access modes. If supplied, *FILENAME* should be a complete and exact filename, including a version number. The device field `OPENFILELST` provides a convenient place to store the open streams. \GENERIC.OPENP is a sample OPENP method that can be used in conjunction with \ADD-OPEN-STREAM and \DELETE-OPEN-STREAM; it consults the device's OPENFILELST.

`(REGISTERFILE` *DEVICE STREAM*`)`

Invoked by `\OPENFILE`, the `REGISTERFILE` method places the newly-opened stream on *DEVICE*'s list of open streams. Note that this need not be an actual list, but might be contained in the device's directory data structure or other convenient place. The function \ADD-OPEN-STREAM, though not the default, is a simple function that adds STREAM to DEVICE's OPENFILELST.

`(UNREGISTERFILE` *DEVICE STREAM*`)`

Invoked by `\CLOSEFILE`, the `UNREGISTERFILE` method removes *STREAM* from *DEVICE*'s list of open streams. The revalidation code also uses this method to silently remove a device's invalid streams. Note that this need not be an actual list, but might be contained in the device's directory data structure.

The function \GENERIC-UNREGISTER-STREAM, though not the default, is a simple function that removes STREAM from DEVICE's OPENFILELST. The revalidation code also uses this method to silently remove a device's invalid streams.

The following methods are invoked for open streams. They are all required:

`(BIN` *STREAM*`)`

Returns the next byte of input from *STREAM*, or takes the appropriate action if at end of file. Unless a device has a good reason not to, it should call `(\EOF.ACTION` *STREAM*`)` at end of file/stream.

The device `BIN` method is actually not used directly. Rather, every stream has a `STRMBINFN` field, which is the function actually applied to do the input. The `STRMBINFN` field could thus be used to fake a specialization of the device differing only in the `BIN` method. However, the typical use of `STRMBINFN` is to temporarily override the device default. In particular, setting a stream's access to `INPUT` or `BOTH` automatically sets the stream's `STRMBINFN` to be the device's `BIN` method; setting access to `NIL` or `OUTPUT` sets the `STRMBINFN` to be an error. This relieves the device's `BIN` method of any need to check the stream's access on every call to `BIN`. Some network streams temporarily set their `STRMBINFN` to be an input eater when they receive a ‘‘clear output’’ command.

Currently, all Interlisp-D streams have bytesize 8, so `BIN` always returns an 8-bit integer.

Calls to the function `BIN` are compiled into the `BIN` opcode, which runs in microcode on some machines if the requirements for it are met. More on this later.

`(BOUT` *STREAM BYTE*`)`

Outputs *BYTE* to *STREAM*. As with `BIN`, this method is not used directly. Rather, every stream has a `STRMBOUTFN` field, which is the function actually applied to do the output. Setting a stream's access to `OUTPUT` or `BOTH` automatically sets the stream's `STRMBOUTFN` to be the device's `BOUT` method.

There exists a `BOUT` opcode, but the design is incomplete.

`(PEEKBIN` *STREAM NOERRORFLG*`)`

Returns the next input byte from *STREAM*, but does not advance the stream pointer. Thus a subsequent `PEEKBIN` or `BIN` will return the same byte. At end of stream, the device should take eof action as with `BIN`, unless *NOERRORFLG* is true, in which case it should return `NIL`.

(READP *STREAM FLG*)

Returns true if input is available from *STREAM*, that is, if a BIN right now would succeed without waiting. Defaults to \GENERIC.READP, which uses EOFP and PEEKBIN.

Roughly speaking, READP is the complement of EOFP for streams that are not arriving in real time. It is interestingly different for network streams, or the keyboard.

FLG is a bit of cruft that not everyone pays attention to, and may be flushed at some point: if *FLG* is NIL, then READP should return NIL if the only input waiting is an end of line character.

(EOFP *STREAM*)

Returns true when *STREAM* is ‘‘at end of file’’, i.e., a BIN would cause an end of file action to occur. Note that for a network stream, it is possible for both EOFP and READP to be false simultaneously, viz. when there is no input waiting (buffered locally), but the remote end of the stream has not indicated that there is no more input.

There are some who call EOFP on streams open only for output. This is a crock; output streams are always at end of file. But to avoid complaints, a device could return T for EOFP on an output stream.

(BLOCKIN *STREAM BUFFER BYTEOFFSET NBYTES*)

Performs bulk input transfer: retrieves the next *NBYTES* bytes from *STREAM* and stores them in successive byte positions in *BUFFER* starting at *BYTEOFFSET*. Defaults to \GENERIC.BINS, which repeatedly calls BIN and \PUTBASEBYTE.

It is almost always the case that a device with a non-trivial BLOCKIN method can be made to be a Buffered device, thereby benefiting from other Buffered operations as well.

(BLOCKOUT *STREAM BUFFER BYTEOFFSET NBYTES*)

Performs bulk output transfer: outputs *NBYTES* bytes to *STREAM*, taking the bytes from *BUFFER* starting at *BYTEOFFSET*. Defaults to \GENERIC.BOUTS, which repeatedly calls \GETBASEBYTE and BOUT.

(FORCEOUTPUT *STREAM WAITFORFINISH*)

Forces to its ultimate destination any output buffered on *STREAM* but not yet sent. *WAITFORFINISH* means that the function should not return until it is confident that the output has reached its destination and been committed. Defaults to NILL, which is reasonable for unbuffered streams.

For example, for a network stream, FORCEOUTPUT sends the current packet being buffered up. For a buffered stream to the disk, FORCEOUTPUT writes out to the disk any ‘‘dirty’’ pages, and makes sure the file is in such a state that if the machine were booted after FORCEOUTPUT returns, that the file could be successfully reopened with no information lost.

(GETFILEINFO *NAME/STREAM ATTRIBUTE DEVICE*)

Returns the value of the specified *ATTRIBUTE* of *NAME/STREAM*, which can be an open Stream or the name of a (closed) file. Returns NIL for attributes it doesn’t know about. It is considered good citizenship, though not absolutely required, to know about the following attributes:

LENGTH	Length of the stream/file in bytes. If the device’s method returns NIL, but the stream is random access, the generic GETFILEINFO code tries the device’s GETEOFPTR method instead.

`SIZE` Length in pages, i.e., `(FOLDHI length BYTESPERPAGE)`.

`CREATIONDATE` Date when the file's contents were created, as a string. The creationdate does not change when a file is copied or renamed, only when it is changed.

`WRITEDATE` Date when the file was written to its current place of storage.

`READDATE` Date when the file was last read.

`ICREATIONDATE`, `IWRITEDATE`, `IREADDATE`
The creation, write and read dates as integers, such as from the function `IDATE`.

`TYPE` Type of the contents: `TEXT` for files that contain only ‘‘text’’ (generally meaning 7-bit ascii), `BINARY` for all others. `NIL` means unknown.

`AUTHOR` Name of the user who created the file (a string).

The following ‘‘generic’’ attributes are generally handled by the generic side of `GETFILEINFO` if the device's `GETFILEINFO` method returns `NIL`:

`EOL` The end of line convention of the stream (`CR`, `CRLF` or `LF`).

`BUFFERS` The number of pagemap buffers for use by the stream (see description of `MAXBUFFERS` field of pagemapped streams).

`ENDOFSTREAMOP` Action to take on any attempt to read beyond the end of file. This is a function of one argument, the stream. The function can cause an error, or return a value, which is interpreted as a value to return from `BIN`. The default `ENDOFSTREAMOP` causes an `END OF FILE` error.

`ACCESS` An atom describing the access mode of the stream (`INPUT`, `OUTPUT`, etc). This is so generic that it is handled before the device's method ever sees it.

`BYTESIZE, OPENBYTESIZE` The size of bytes transmitted on the stream. Always 8 these days.

`(SETFILEINFO` *NAME/STREAM ATTRIBUTE VALUE DEVICE*`)`

Sets the value of the specified *ATTRIBUTE* of *NAME/STREAM* to be *VALUE*. Returns `T` if successful, `NIL` if unsuccessful, or for attributes it doesn't know about.

It is not generally required that `SETFILEINFO` recognize any attributes at all—`NILL` is a perfectly good filler for this slot. Most devices recognize no more than `TYPE` and `CREATIONDATE` (`ICREATIONDATE`), and even those are not very important, as most applications set those attributes in the *PARAMETERS* argument to `OPENFILE` when creating a file.

ATTRIBUTE = `LENGTH` implies actually truncating (or lengthening) the file; however, the `SETFILEINFO` need not handle this itself—if it returns `NIL`, then the generic file system will attempt to use the `SETEOFPTR` method instead.

The following operations are only required of random access streams. They default to the function `\IS.NOT.RANDACCESSP`, which causes a ‘‘Stream is not randaccessp’’ error when called.

`(GETFILEPTR` *STREAM*`)`

Returns the current file pointer (byte position) in *STREAM*. The file pointer is zero when the stream is opened (except for *ACCESS* = `APPEND`), and is incremented by one for each byte read.

Although this operation is only absolutely required for random access streams, it is desirable to supply it for other streams where possible. For example, when reading a file sequentially through PupFtp, the stream can count the bytes as they go by and thus give an accurate value for GETFILEPTR. If a stream has no idea at all of position, it can make its GETFILEPTR be the function ZERO and thereby at least avoid breaks from code that calls GETFILEPTR carelessly.

(GETEOFPTR *STREAM*)

Returns the file pointer of the end of *STREAM*, i.e., the file pointer that GETFILEPTR would return after the last byte of *STREAM* is read. Same as the LENGTH attribute for a stream that represents a file. Of course, non-random access streams may have no idea where the end is, and causing a non-randaccessp error is perfectly acceptable.

(SETFILEPTR *STREAM* *BYTENUMBER*)

Sets the file pointer of *STREAM* to be *BYTENUMBER*. The special value *BYTENUMBER* = ÿ1 means the end of the stream; other negative values are illegal.

SETFILEPTR beyond the end of the stream is permissible, but it has no immediate effect beyond changing the logical file pointer. Attempting to then BIN causes an EOF error. Attempting to BOUT (for a file open for write) should extend the file, so that its eof is immediately beyond the newly BOUTed byte.

As with GETFILEPTR, there is no requirement that this work on non-random access streams, and it may be completely impossible on some of them. However, for those non-random access streams that perform GETFILEPTR, it is possible to fake SETFILEPTR for values larger than the current file pointer by skipping some number of bytes in the file, e.g., by performing (RPTQ (DIFFERENCE *BYTENUMBER* (GETFILEPTR *STREAM*)) (BIN *STREAM*)). There are some applications for which forward SETFILEPTR is all the random access that is actually required, so it is nice to be able to accommodate such applications.

(BACKFILEPTR *STREAM*)

Backs up the file pointer in *STREAM* by one byte. Functionally the same as (SETFILEPTR *STREAM* (SUB1 (GETFILEPTR *STREAM*)), but may be possible on non-random access streams by maintaining a one-character buffer, which is all the backing up this operation is formally required to perform. I believe the main use for this is in READ, which needs to back up the stream one character when, for example, it reads a break character terminating an atom.

(SETEOFPTR *STREAM* *LENGTH*)

Changes the length of *STREAM* to be *LENGTH*, i.e., ‘‘sets’’ its end of file pointer. This may require lengthening or truncating the file. Used by the function \SETEOFPTR and by SETFILEINFO for attribute LENGTH when the device’s SETFILEINFO method doesn’t handle it.

The following three fields are place holders for possible future extensions. These fields are not currently used at all:

(LASTC *STREAM*)

Returns the last character read from *STREAM*, i.e., the last byte that was BINed, as a character. LASTC is currently implemented via BACKFILEPTR.

(FREEPAGECOUNT *HOST/DIR* *DEVICE*)

Intended use is to return the number of free pages on *HOST/DIR*. May be folded into a general GET/SET device/directory info operation.

(`MAKEDIRECTORY` *HOST/DIR DEVICE*)

Intended use is to create a new directory *HOST/DIR*.

The remaining fields in the `FDEV` are for buffered and page-mapped streams, and are ignored for non-buffered devices. These fields are described in separate sections.

Streams

The following fields are used by all streams:

`DEVICE` — Pointer to this stream's `FDEV`.

`FULLFILENAME` — ''Full'' name by which this file is known to the user. Should be an uppercase litatom, fully qualified so that giving the same name back to the file system should produce the same file (to the extent that the device can support such uniqueness). Is `NIL` for unnamed streams.

`FULLNAME` — Access field. Is the same as `FULLFILENAME`, unless that is `NIL`, in which case it is the stream itself. This avoids the circularity that would result if the FULLFILENAME field contained the stream datum.

`NAMEDP` — Access field. Is `T` if the streams is named, i.e., its `FULLFILENAME` is non-`NIL`.

`ACCESSBITS` — Contains a numeric code describing what access mode the file is open for: there are read, write and append bits. This field is usually accessed indirectly via the `ACCESS` field. However, there are macros for referring to particular types of access using more efficient bit test operations:

- (`OPENED` *STREAM*) — `ACCESS` is not `NIL`.
- (`READABLE` *STREAM*) — Read bit is on: `ACCESS` is `INPUT` or `BOTH`.
- (`READONLY` *STREAM*) — Only the read bit is on: `ACCESS` is `INPUT`.
- (`APPENDABLE` *STREAM*) — Append bit is on: `ACCESS` is `OUTPUT`, `BOTH` or `APPEND`.
- (`APPENDONLY` *STREAM*) — Only the append bit is on: `ACCESS` is `APPEND`.
- (`DIRTYABLE` *STREAM*) — Append or write bit is on: `ACCESS` is `OUTPUT`, `BOTH` or `APPEND`. Yes, this is operationally the same as APPENDABLE, given the four possible values of ACCESS.
- (`OVERWRITEABLE` *STREAM*) — Write bit is on: `ACCESS` is `OUTPUT` or `BOTH`.
- (`WRITEABLE` *STREAM*) — Write bit is on, or append bit is on and file is at EOF. Avoid using this one, it's a little strange.

`ACCESS` — Access field for referring to the `ACCESSBITS` field symbolically. Its value is one of the legal values of the *ACCESS* argument to `OPENFILE`: `INPUT`, `OUTPUT`, `BOTH`, `APPEND`; or `NIL` when the stream is closed. Replacing this field has the side effect of setting the

BINABLE, BOUTABLE, STRMBINFN and STRMBOUTFN fields appropriately (from the corresponding device fields, or to values consistent with no access).

USERCLOSEABLE — Flag, true if the stream can be closed by CLOSEF. Default is T, but is NIL for such things as dribble files and the terminal.

USERVISIBLE — Flag, true if the stream is to be listed in the result of (OPENP). Default is T, but is NIL for such things as dribble files and the terminal.

BINABLE — True if BIN microcode can be used. Normally set automatically from FDBINABLE when input access is set.

BOUTABLE — True if BOUT microcode can be used. Normally set automatically from FDBOUTABLE when output access is set.

EXTENDABLE — True if BOUT can extend the buffer when COFFSET reaches CBUFSIZE. Obsolete.

STRMBINFN — Function called by BIN. This is normally set indirectly as a side effect of setting the ACCESS field. Setting ACCESS to an input access (INPUT or BOTH) sets the STRMBINFN to be the stream's device's BIN method. Setting to any other access sets the STRMBINFN to be a ‘‘file not open’’ trap.

STRMBOUTFN — Function called by BOUT. As with STRMBINFN, this is normally set indirectly (from the device's BOUT method) as a side effect of setting the ACCESS field.

OUTCHARFN — Function called to output a single byte. This is like STRMBOUTFN, except for being one level higher: it is intended for text output. Hence, this function should convert (CHARCODE EOL) into the stream's actual end of line sequence, and should adjust CHARPOSITION appropriately before invoking the stream's STRMBOUTFN to actually put the character. Defaults to \FILEOUTCHARFN. The OUTCHARFN for the display additionally worries about such things as ECHOCONTROL.

CHARPOSITION — Current horizontal character position in the stream. Incremented (and reset to zero) by OUTCHARFN. Used by the function POSITION.

LINELENGTH — Maximum line length of the stream, in characters. Used by the function LINELENGTH. Defaults (at creation time) to the value of the global variable FILELINELENGTH.

EOLCONVENTION — The stream's end of line convention: the manner in which ‘‘end of line’’ is encoded on this stream. That is, output of an end of line (function TERPRI) produces the stream's end of line sequence, and on input, the stream's end of line sequence is converted to (CHARCODE EOL) by READC. This is not necessarily the same as the way that end of line is encoded in the actual file written by, say, a file server. For example, Lisp might open a stream to a Tenex file server with EOLCONVENTION of CR, while the server might choose to take each of the CRs in the stream and actually store a CR, LF sequence in the physical file.

The convention is encoded as a two-bit field; the constants CR.EOLC, LF.EOLC, CRLF.EOLC can be used to refer to the currently known values symbolically. Default in Interlisp-D is CR.EOLC.

`ENDOFSTREAMOP` Function of one argument (the stream) called when an attempt to read beyond the end of file occurs. If this function returns something, it should be interpreted as a value to return from `BIN` (the value `T` is currently prohibited). Defaults to `\EOSERROR`, which causes an `END OF FILE` error.

`VALIDATION` Pointer field, some compact encoding of the state of the file such that if the file's content changes, the `VALIDATION` changes. The file's `ICREATIONDATE` attribute usually works well enough. The only use for this field is to check whether the file changed over `LOGOUT`, etc.—if the `VALIDATION` of the stream returned from `REOPENFILE` is `EQUAL` to the `VALIDATION` of the stream open before `LOGOUT`, the stream is assumed to be unchnaged. This will probably be the sole concern of the device when we go to multiple streams per file.

`BYTESIZE` Byte size of the file, i.e., what `BIN` and `BOUT` traffic in. Defaults to 8. This field is not used by many; there are probably a lot of things that won't work if the byte size is not 8.

`OTHERPROPS` List in property list format used by the function `STREAMPROP`. Analogous to `WINDOWPROP`, etc.

`IMAGEOPS` Image operations vector (object of type `IMAGEOPS`) for use of device-independent graphics operations, such as `DSPXPOSITION`, `DSPFONT`. Defaults to `\NOIMAGEOPS`, a vector suitably defined for non-display devices. See the implementors' manual chapter **Device-Independent Graphics**.

`IMAGEDATA` Device-dependent data for use by `IMAGEOPS`.

`REVALIDATEFLG` Flag. The standard use of this flag is to solve a problem with correctly maintaining the creation date. The problem is that the definition of ‘‘creation date’’ is that the creation date changes whenever the contents of the file change. If followed literally, this would mean, for example, that ever time you wrote out a page of a {`DSK`} file, you would also have to rewrite its leader page with a new creation date. However, it suffices in practice to only change the creation date when it would matter, i.e., when there would be any possibility of some agent other than the currently running Lisp to see the change. Usually, this means the only time to worry about is when the Lisp vmem is saved and a file that was open before the save is written to again afterwards.

Thus, the use of this flag (for those devices that care) is as follows: the device's `BEFORExxx` events set this flag true for any streams open on the device. Then, whenever the device is about to do something that would change the file's content, e.g., write out a new page, it first tests `REVALIDATEFLG`. If the flag is true, it updates the file's creation date and clears the flag.

`NONDEFAULTDATEFLG` Flag. Standard use is in conjunction with `REVALIDATEFLG`, to mark a file that was opened in a way that the user constrained the creation date of the file (e.g., the *PARAMETERS* argument to `OPENFILE` included an explicit creation date, or the option `DON'T.CHANGE.DATE`).

`F1`, `F2`, `F3`, `F4`, `F5` Pointer fields for private use by the stream, to maintain stream-specific state of concern only to the device. Stream clients that wish to hang information on a stream without regard to what kind of stream it is should use the function `STREAMPROP`.

`FW6`, `FW7`, `FW8`, `FW9` 16-bit word fields for private use by the stream.

DIRTYBITS — Obsolete.

EXTRASTREAMOP — ?

Buffered Streams

Buffered streams are ones that constrain themselves to obey a set of conventions that make it easy for an agent (e.g., microcode) to perform input or output on the stream without knowing about the details of the stream's physical i/o. The stream maintains a ‘‘current buffer’’ and two indices into that buffer, the offset of the next byte, and the offset of the end of the buffer. As long as the former index is less than the latter, the stream guarantees that the bytes in the buffer between those indices are the true contents of the file/stream starting at the current file pointer. Advancing the first index effectively advances the file pointer. When it reaches the second index, a stream-specific operation is called to ‘‘refill’’ the buffer.

The following fields are used by buffered streams:

COFFSET — Byte offset in the buffer CBUFPTR of the next BIN or BOUT.

CBUFSIZE — ‘‘Size’’ of the current buffer, i.e., byte offset that is one beyond the last byte.

CBUFMAXSIZE — For output, the maximum size the buffer can be written to. If COFFSET reaches CBUFSIZE, but CBUFSIZE is less than CBUFMAXSIZE, then the buffer can be extended.

CBUFPTR — Pointer to current buffer. Must be valid if COFFSET is less than CBUFSIZE and BINABLE or BOUTABLE is true. It is not necessary that this ‘‘buffer’’ be anything other than some chunk of memory, a portion of which contains interesting data. Thus, the bytes from offset COFFSET to CBUFSIZE must be valid, but COFFSET need not start at zero, nor need CBUFSIZE or CBUFMAXSIZE coincide with the end of the underlying structure.

CBUFDIRTY — Flag, true if current buffer has been written to.

In general, the device has sole responsibility for setting CBUFSIZE, CBUFMAXSIZE, and CBUFPTR; generic code does not touch those. The fields COFFSET and CBUFDIRTY can be changed by generic stream clients as well as by device-specific code. For example, code that simulates a BIN increments COFFSET; code that writes directly to the stream's buffer sets CBUFDIRTY true.

The following methods are defined for devices implementing buffered streams:

(GETNEXTBUFFER *STREAM* *WHATFOR* *NOERRORFLG*) [Device method]

Called when *STREAM* needs to have its buffer fixed, i.e., the state of *STREAM* is such that BIN (*WHATFOR* = READ) or BOUT (*WHATFOR* = WRITE) cannot proceed. This method should do whatever is necessary to allow the operation to proceed. This typically includes disposing of the current buffer somehow (if GETNEXTBUFFER was invoked because the buffer was exhausted), and fetching a new buffer consistent with *STREAM*'s current position.

In the case of *WHATFOR* = READ, GETNEXTBUFFER returns T on success, i.e., if *STREAM* is not at end of file. When *STREAM* is at end of file, GETNEXTBUFFER should take standard end of stream action, returning whatever \EOF.ACTION returns (if anything). However, if *NOERRORFLG* is true, GETNEXTBUFFER should just return NIL immediately.

(RELEASEBUFFER *STREAM BUFFER*) [Device method]

Performs any device-specific operation required when *BUFFER*, which is the current value of *STREAM*'s CBUFPTR field, is ‘‘released’’ (when the CBUFPTR field is replaced). This is used so that different pagemap-like devices can share certain code. For example, in the case of pagemapped streams, RELEASEBUFFER marks the buffer dirty in the case that the stream's CBUFDIRTY field has been set.

This method is not currently used.

The functions \BUFFERED.BIN, \BUFFERED.PEEKBIN, \BUFFERED.BOUT, \BUFFERED.BINS and \BUFFERED.BOUTS are supplied for use by buffered streams; they are standardly used to implement the BIN, PEEKBIN, BOUT, BLOCKIN and BLOCKOUT device methods. In addition, the function COPYBYTES, when presented with a source stream that is buffered, utilizes the GETNEXTBUFFER method to efficiently copy bytes to the destination a buffer-full at a time.

Pagemapped Streams

Pagemapped streams are a particular kind of random access Buffered stream that buffers its data in units of pages. The device provides methods that read or write data in units of pages, while system-supplied Pagemapped functions handle the responsibilities of a Buffered stream, as well as managing the file pointer for random access. In general, a stream can have several pages of a file buffered at a time, allowing the code to make some effort to make efficient use of multi-paged transfers where applicable.

To create a pagemapped device, create an FDEV, fill in the necessary private fields, then call the following function:

(\MAKE.PMAP.DEVICE *DEVICE*) [Function]

Fills in fields in the device appropriate for pagemapped devices, and returns the updated device. The fields it fills are the flag fields FDBINABLE, FDBOUTABLE, RESETABLE, RANDOMACCESSP, PAGEMAPPED, BUFFERED (all true), and the methods BIN, BOUT, PEEKBIN, BLOCKIN, BLOCKOUT, READP, EOFP, GETFILEPTR, BACKFILEPTR, SETFILEPTR, GETEOFPTR, SETEOFPTR, GETNEXTBUFFER and FORCEOUTPUT.

A Pagemapped device is required to supply the following methods (in addition to those required of all devices and not filled in by \MAKE.PMAP.DEVICE):

(READPAGES *STREAM FIRSTPAGE# BUFFERS*) [Device method]

Causes pages of *STREAM* to be read into *BUFFERS*. The first page read is *FIRSTPAGE#* (zero for the first page of the file). *BUFFERS* is either a single page-sized buffer (a VMEMPAGEP), in which case exactly one page is read, or it is a list of such buffers. READPAGES returns the total number of bytes read. If the last page read is not a full page, READPAGES should zero out the rest of its buffer. READPAGES can assume that the buffers are page-aligned, although they need not be consecutive.

(WRITEPAGES *STREAM FIRSTPAGE# BUFFERS*) [Device method]

Writes data from *BUFFERS* out to *STREAM*. The first page written is *FIRSTPAGE#*. *BUFFERS* is as with READPAGES.

Neither READPAGES nor WRITEPAGES affects *STREAM*'s file pointer or end of file; those are managed by higher-level pagemapped routines. WRITEPAGES might, however, want to look at *STREAM*'s EPAGE and EOFFSET fields if it needs to take any special action around the end of the file. It is possible, for no particularly good reason, for

READPAGES to get called for a page beyond the end of file; in fact, this standardly happens when writing a new file. The READPAGES method in this case should just clear the buffer and return zero.

(TRUNCATEFILE *STREAM PAGE# OFFSET*) [Device method]

Truncates *STREAM* so that its end of file is *PAGE#*, *OFFSET*, which should be defaulted to *STREAM*'s EPAGE and EOFFSET. Can be used to either shorten or lengthen a file; if lengthening, the file should be padded with nulls. Used by \PAGED.SETEOFPTR and \PAGED.FORCEOUTPUT. As of this writing there are still bugs in this code in certain funny cases, such as when you SETFILEPTR beyond eof and then BOUT.

The following fields of a stream are meaningful for a pagemapped device. The generic pagemapped codes maintain them as operations on the file are performed, but they should all be initialized appropriately by the device's OPENFILE method:

CPAGE — For pagemapped streams, the current page position in the stream. Together with COFFSET, this constitutes the stream's file pointer. The device's OPENFILE method should set CPAGE and COFFSET to zero, except for files opened with access APPEND, in which case they should be set to the end of file.

EPAGE, EOFFSET — For pagemapped files, the page and byte offset of the end of file. Note that this is the *logical* end of the file; it need have nothing to do with the physical end of file, except that when a file is closed, the device should see to it that its logical and physical EOFs are the same (normally seen to by the TRUNCATEFILE inside of \CLEARMAP, below). In fact, as a typical file is being written, EPAGE tends to stay several pages ahead of the physical end of file by virtue of the fact that pages are being buffered before being written out.

BUFFS — For pagemapped streams, a pointer to the stream's BUFFER chain. Initially NIL (no buffers allocated). The device usually has no direct interest in this field.

MAXBUFFERS — For pagemapped streams, the maximum number of buffers desired in the stream's BUFFS chain. If the code needs another buffer and there are already MAXBUFFERS buffers, it will try to recycle the least recently referenced buffer. Defaults to \STREAM.DEFAULT.MAXBUFFERS. The user can change this field for an open stream by calling SETFILEINFO with attribute BUFFERS.

MULTIBUFFERHINT — Flag. For pagemapped streams, is a hint to the pagemap code that the device prefers to transfer data more than one buffer at a time. If this flag is true, the pagemap code tries to write out (WRITEPAGES) more than one buffer at a time when the opportunity arises. A similar improvement is planned, but not implemented, for reading multiple buffers at a time.

The following functions are of use for pagemapped devices:

(\PAGED.FORCEOUTPUT *STREAM WAITFORFINISH*) [Function]

This function implements the FORCEOUTPUT method for pagemapped streams: it causes any dirty pages to be written out (using WRITEPAGES), then calls the TRUNCATEFILE method to set the end of file.

This function is normally installed as the FORCEOUTPUT method by the function \MAKE.PMAP.DEVICE. However, the device can override this default (by supplying its own function in that field), in which case it might want to call the function \PAGED.FORCEOUTPUT explicitly as part of its more comprehensive FORCEOUTPUT method.

There is an unpleasantness in the implementation of pagemapped devices that stems from the fact that originally all devices (the few that existed in the distant past) were made to support the PMAP package, a means whereby a programmer could get direct access to the buffers of a file, much as one can with the PMAP JSYS in Tenex. As a result, the buffers used by pagemapped streams are set up in a special manner so that the garbage collector can tell when the user no longer has access to a PMAP buffer. The PMAP package is being phased out.

This is all exceedingly crufty, and is of little concern to the device implementer, except for the fact that it requires that the buffers be explicitly released when a stream is closed; the buffers are not automatically collected when the stream is dropped.

(FORGETPAGES *STREAM* *FROMPAGE* *TOPAGE*) [Function]

> ‘‘Forgets’’ pages *FROMPAGE* thru *TOPAGE* of *STREAM*; i.e., removes those pages from the set of pages being currently buffered, and frees the buffers they were occupying. If *FROMPAGE* = *TOPAGE* = NIL, forgets all pages, and releases all of *STREAM*’s buffers.

(\CLEARMAP *STREAM*) [Function]

> Performs a FORCEOUTPUT (if *STREAM* is open for output) followed by a FORGETPAGES. This is the standard action that should be taken by a pagemapped stream’s CLOSEFILE method.

Directory Enumeration

This section describes how directory enumeration works—what you need to know in order to implement the GENERATEFILES device method, and what you need to know as a programmer trying to enumerate a directory via anything more elaborate than the function DIRECTORY.

The general idea is that the directory enumeration code is given a pattern, and it returns a generator that, each time it is poked, returns another file name matching the pattern. In addition, the generator provides a handle for getting file attributes of each enumerated file. This second handle is important for efficiency: although one could just take the file name given by the enumerator and pass it to GETFILEINFO, the device, in the course of enumeration, usually has its fingers on the file closely enough that it need not perform the second directory lookup that a GETFILEINFO out of the blue would require. The caller of the directory enumeration code specifies ahead of time which, if any, attributes will be required (a necessity for most file server implementations).

Information for device implementors. A *file generator* is an object represented as a list described by the record FILEGENOBJ, exported from FILEIO:

```
(RECORD FILEGENOBJ (NEXTFILEFN FILEINFOFN . GENFILESTATE))
```

NEXTFILEFN and FILEINFOFN are functions of the device’s choosing that when called will return the next file, and attributes for that file. GENFILESTATE is arbitrary state maintained by the generator. With that as background, here are the pieces of directory enumeration:

(GENERATEFILES *DEVICE* *PATTERN* *DESIREDPROPS* *OPTIONS*) [Device method]

> Returns a generator that enumerates files matching *PATTERN*, which is a string that has host and directories suitably filled in from defaults, and may contain the pattern character ‘‘*’’ to match an arbitrary number of characters. *DESIREDPROPS* is a list of file attributes that may be requested during the enumeration; they must be valid *ATTRIBUTE* arguments to GETFILEINFO. *OPTIONS* is a list of options to the enumeration, chosen from among the following:

`SORT`	The files should be enumerated in sorted order. If this option is not specified, the device is free to enumerate files in any convenient order. There is some question as to whether files should be enumerated lowest version first (as IFS's do) or highest version first (as Twenex does). I prefer the latter, but given servers that do the former, we currently make no requirement about version order.
`RESETLST`	Informs the enumerator that the enumeration context is surrounded by a `RESETLST`, so that it may perform `RESETSAVE`s to clean up after itself if the enumeration is aborted. Cleaning up can be a very messy business without this information about the scope of the enumeration, so all callers of `\GENERATEFILES` are strongly encouraged to provide it.

`GENERATEFILES` should return a file generator with a suitable `NEXTFILEFN` and `FILEINFOFN`.

Fine point about missing fields in the pattern: null fields in *PATTERN* match only files for which the corresponding field is null. A null version is interpreted as highest. Thus,

`DIR *` = `DIR *.*` = `DIR *.*;*` enumerates everything.

`DIR *.` = `DIR *.;*` enumerates all versions of files with null extension.

`DIR *.;` enumerates highest version of files with null extension.

`DIR *.*;` enumerates highest version of everything.

It is difficult for some devices to enumerate only highest version of files; there are several devices in the system that treat a null version the same as version `*`. However, every device should try its best. With some work, any device that can enumerate all versions can enumerate just highest version if it enumerates in sorted order and uses perhaps a little lookahead to assure that any name it returns is the one of highest version.

`(NEXTFILEFN` *GENFILESTATE NAMEONLY*`)` [File Generator Component]

Generates the next file, returning its name as a string, or `NIL` if the generator is exhausted. *GENFILESTATE* is the state component of the file generator returned from `GENERATEFILES`. *NAMEONLY* means that the caller is only interested in the file's `Name.Ext` fields, not the full file name (and no more than one version of the file need be enumerated); however, it is always permissible to return the full file name. The *NAMEONLY* option is used by `SPELLFILE`.

`(FILEINFOFN` *GENFILESTATE ATTRIBUTE*`)` [File Generator Component]

Returns the value of the *ATTRIBUTE* property of the file most recently generated by the `NEXTFILEFN`, i.e., effectively `(GETFILEINFO` latest-name *ATTRIBUTE*`)`, but hopefully much faster. *ATTRIBUTE* must have been a member of the *DESIREDPROPS* argument to `GENERATEFILES`.

Not all device implementors are enthused about implementing a pattern matcher for file names. The following functions are provided to help out:

`(DIRECTORY.MATCH.SETUP` *PATTERN*`)` [Function]

Accepts as *PATTERN* a file name string such as passed to `GENERATEFILES`. Returns an object suitable as a filter to `DIRECTORY.MATCH`.

`(DIRECTORY.MATCH` *FILTER TESTNAME*`)` [Function]

Matches *TESTNAME*, a file name, against *FILTER*, the object returned from DIRECTORY.MATCH.SETUP. Returns true if *TESTNAME* matches the pattern, false if not. The match is case-insensitive.

(\NULLFILEGENERATOR) [Function]

Returns a file generator that produces no files.

(\GENERATENOFILES *DEVICE PATTERN DESIREDPROPS OPTIONS*) [Function]

Returns a ''stupid'' file generator for devices that don't know how to enumerate in general. If *PATTERN* contains no wildcards, but names a file that is INFILEP, then the generator produces exactly that file. If *PATTERN* contains a wildcard in the version field, it uses GETFILENAME to laboriously generate all the versions of the file. In all other cases, \GENERATENOFILES returns a null file generator.

Information for clients of device enumeration. The following functions make up the ''public'' interface to directory enumeration:

(\GENERATEFILES *PATTERN DESIREDPROPS OPTIONS*) [Function]

Returns a file generator object for enumerating the files matching *PATTERN*. *PATTERN* is expanded by adding the default host and/or directory if appropriate. See description of the GENERATEFILES method for description of *DESIREDPROPS* and *OPTIONS*.

(\GENERATENEXTFILE *GENERATOR NAMEONLY*) [Function]

Returns the next file, as a string. *GENERATOR* is the object returned from \GENERATEFILES; *NAMEONLY* indicates caller does not require that the full name be returned, but that the name and extension are sufficient.

(\GENERATEFILEINFO *GENERATOR ATTRIBUTE*) [Function]

Returns the value of the *ATTRIBUTE* property of the file most recently generated by \GENERATENEXTFILE, i.e., effectively (GETFILEINFO latest-name *ATTRIBUTE*). *ATTRIBUTE* must have been a member of the *DESIREDPROPS* argument to \GENERATEFILES.

(DIRECTORY.FILL.PATTERN *PATTERN DEFAULTEXT DEFAULTVERS*) [Function]

This function is used to fill in defaults in *PATTERN* before passing it to \GENERATEFILES. If *PATTERN* does not include an extension or version, but those fields are not explicitly omitted (e.g., ''FOO'', but not ''FOO.''; ''FOO.BAR'', but not ''FOO.BAR;''), they are filled in with *DEFAULTEXT* and *DEFAULTVERS*, which themselves default to ''*''. This function is used by the DIR command, and should probably be used by any code that takes a user-supplied pattern and enumerates files from it.

Streams and File Devices

Edited 30 Nov 84, van Melle

An Interlisp Stream is an object of datatype STREAM that is capable of performing, at the least, sequential input and/or output of bytes. Some streams can do much more. Streams are used for access to open files, for writing to the display, for chatting to remote hosts, and whatever other uses people come up with. This document describes how one goes about defining a new device, the meanings of the record fields of the STREAM and FDEV datatypes, and anything else that seemed relevant at the time.

The implementation of Streams is strongly object-oriented. Every STREAM has a pointer to a device (the datatype FDEV), which contains a vector of functions to be called when certain operations are required of the stream. There can be many streams with the same device. In the object-oriented terms of LOOPS, one can think of the device as a *class*, which provides a set of *methods* that implement class operations, and the streams as *instances*. Devices and streams also have local state, which might be thought of as class and instance variables. Declarations for STREAM and FDEV can be obtained by loading EXPORTS.ALL.

OPENSTREAM, CLOSEF, FORCEOUTPUT, READP, EOFP, GETEOFPTR, GETFILEINFO, SETFILEINFO, DIRECTORY, COPYBYTES, DELFILE, RENAMEFILE, FULLNAME are some of the Lisp functions called by the programmer that ultimately turn into operations at the device level. The descriptions that follow sometimes allude to these functions, and knowledge of how they operate may occasionally give the reader additional clues as to how the device operations work.

Typically, some part of the operation is handled by the ‘‘generic’’ file system code, which then calls on the device to handle that part of the operation that is device-specific. For example, the function OPENFILE takes the name of the file it is to open and fills in host and directory defaults, and decides which device handles such a file. It then calls on the particular device to actually open the file. After the file is opened, the generic file system code registers the file on (OPENP). As another example, operations involving open streams first coerce non-streams (e.g. filenames) to open streams before calling the device-specific operation.

Devices

A device is an object of type FDEV (so named for historical reasons: ‘‘File Device’’). The standard way to define a new device is to create such an object, by performing (**create** FDEV --), and then pass the newly created FDEV to the function \DEFINEDEVICE. \DEFINEDEVICE is the way a device ‘‘announces’’ itself to the generic file system.

(\DEFINEDEVICE *NAME* *DEV*) [Function]

Installs device *DEV*, giving it the name *NAME*. *NAME* must be an uppercase litatom. The generic file system code makes use of the name to locate the device that is willing to deal with files whose full name begins {*NAME*}. It is permissible to have more than one name map to the same device; this effectively provides device synonyms. Devices are encouraged, however, to always create file names using the canonical device name, independent of what name was passed in.

NAME can be NIL, in which case the generic file system code never consults the device directly. However, its EVENTFN is still run around Lisp exits, and it can be used as the device for a stream created by nonstandard methods.

If a device never wants to be invoked by name, and has no interesting EVENTFN or HOSTNAMEP methods, then there is no need to ever register it with \DEFINEDEVICE.

`(\REMOVEDEVICE` *DEV*`)` [Function]

Removes device *DEV* from the list of known devices, as well as any name that maps to that device.

`(\GETDEVICEFROMNAME` *NAME NOERROR DONTCREATE*`)` [Function]

Returns the device associated with *NAME*. *NAME* can be a litatom or string; it is coerced to uppercase, and if it begins with an open brace, is assumed to be a file name, from which the host name is extracted. If no such device is known, attempts to find one by polling the `HOSTNAMEP` methods of all known devices (see below); if a device is still not found, causes a `FILE NOT FOUND` error unless *NOERROR* is true. If *DONTCREATE* is true, it never attempts to create a device, just returns an existing device if there is one, `NIL` otherwise.

The fields of an `FDEV` are divided up into informational fields and ‘‘methods’’.

`DEVICENAME` A pointer field, the name of the device, standardly a litatom. Use of this field is largely up to the device, but it is usually selected to be the name that appears inside braces in filenames opened on this device. For devices that do not support the notion of named files, `DEVICENAME` can be anything that the implementor cares to use for debugging assistance.

`RESETABLE` A flag, true if `(SETFILEPTR` stream `0)` can be performed. Currently unused.

`RANDOMACCESSP` True if the stream is randomly accessable, i.e., if `SETFILEPTR` works on this kind of stream.

`NODIRECTORIES` True if files opened on this device do not (usually) have a directory as part of their name. The principal use for this is by the `CONN` command, which will not try to connect to the user’s home directory if given a host only, e.g., `CONN {DSK}`.

`BUFFERED` True if streams of this sort are buffered in a manner compatible with the microcoded versions of `BIN` and `BOUT`. More specifically, `BUFFERED` implies that the device implements the `GETNEXTBUFFER` method. See description of buffered streams.

`PAGEMAPPED` True if this stream is implemented by the pagemapped functions. All pagemapped streams are also buffered, so if this flag is true, so should be `BUFFERED`. See description of pagemapped streams.

`FDBINABLE` True if streams on this device obey the rules for microcoded `BIN` whenever such stream is open for input access.

`FDBOUTABLE` True if streams on this device obey the rules for microcoded `BOUT` whenever such stream is open for output access. Currently unused, as the spec needs revision.

`FDEXTENDABLE` Special kind of `FDBOUTABLE`. Currently unused, as the spec needs revision.

`DEVICEINFO` A pointer to arbitrary device-specific information. The standard use for this is to hold local state specific to one of several similar devices that share methods. For example, the Dolphin disk provides a separate `FDEV` for each partition of the machine; the `DEVICEINFO` field of each has pointers to the partition’s directory and other information specific to files on that partition only.

The following fields are all pointer fields, and contain functions for implementing various device operations. Not all devices need have all fields filled in; the required ones are listed first and so indicated. Some ‘‘required’’ fields have defaults specified in the `FDEV` (or `STREAM`) record declaration, so the implementor need not explicitly fill those fields if the default is reasonable. Each field is presented with its arguments, in the style of a function definition; of course, it is the contents of the field, not the field name, that is the function. Using object-oriented terminology, the occupants of these fields are referred to as ‘‘methods’’. For example, ‘‘the `BIN` method’’ means ‘‘the function that occupies the `BIN` field’’.

One of the arguments to each method is usually either the device itself, or a stream open on the device, so that the device (and hence its `DEVICEINFO`) is usually accessible to all these functions. Arguments that are file names or patterns or pieces of file names can be either litatoms or strings, and already have their host and/or directory parts appropriately filled in from the connected directory defaults. The device may assume that the host field of the file name is indeed a name that the device has said it implements (see `HOSTNAMEP`). ‘‘Full’’ file names returned by these functions (or stored in the `FULLFILENAME` field of a stream) should be litatoms, and at least in the current implementation should be all uppercase.

Fields required of every device:

(`HOSTNAMEP` *HOSTNAME DEVICE*)

Called by the generic file system code when presented with a host name for which there is as yet no device defined. The function should return non-`NIL` if it ‘‘recognizes’’ *HOSTNAME*. There are two ways in which this method is invoked:

(1) To obtain a device for *HOSTNAME*, for example, so that a file can be opened on it. In this case, *DEVICE* is an already defined device (the one whose `HOSTNAMEP` method is being called), and the function should return either a new device, or `T`, meaning it is willing to take responsibility for this host name as well as any previous name under which the device was registered. In either case, the caller will install the returned device, or *DEVICE* if value was `T`, as the device to which *HOSTNAME* maps.

(2) As a pure predicate. In this case, *DEVICE* is `NIL`, and the function need only return `T` or `NIL`, indicating whether it believes that *HOSTNAME* is the name of a host.

In practice, the `HOSTNAMEP` method need only take care of the first case, since that also takes care of the second case. The second case is provided so that the device need not be created until there is an actual use for it, should the device wish to avoid unnecessary work. In practice it is rare that anyone tests a host name without subsequently needing to have the device created in full.

There are basically three kinds of devices in the system as distinguished by their `HOSTNAMEP` methods.

(1) Predefined devices with exactly one name, or strictly internal devices with no notion of name. For example, the `CORE` device always exists, and has exactly one name; the `SPP` device (a network byte stream) has no name (it supports no files directly). Such devices have a `HOSTNAMEP` method of `NILL`—the only name they ever go by is the one they gave to `\DEFINEDEVICE`, if any. This is the default.

(2) Devices that don’t know ahead of time what their name will be, but for which there might be many incarnations. This is the model for remote file servers. The standard way of handling this case is to define a dummy device that has only a `HOSTNAMEP` method, and no name. When the `HOSTNAMEP` method gets called with a name that the device knows it can service, it creates a device by that name. If given a name that is a synonym of another name, it might just return the existing device of the canonical name (using `\GETDEVICEFROMNAME` to find the right device).

In either case, the HOSTNAMEP method of the new device is usually NILL—the original device is the only one that worries about creating new instances of this class of device.

(3) Like (2), but all the different names are handled by a single device, which takes care internally of the multiplexing among, say, different remote hosts. HOSTNAMEP returns T in this case. This is usually clumsier than (3), so discouraged.

(EVENTFN *DEVICE EVENT*)

Called around Lisp exits, to allow the device to do any necessary cleaning up, clearing of caches, disconnects with remote hosts, etc. *EVENT* is one of the following litatoms: BEFORELOGOUT, BEFORESYSOUT, BEFOREMAKESYS, BEFORESAVEVM, AFTERLOGOUT, AFTERSYSOUT, AFTERMAKESYS, AFTERSAVEVM, AFTERDOSYSOUT, AFTERDOMAKESYS, AFTERDOSAVEVM. The AFTERxxx events are all run when Lisp is booted from a memory image that resulted from a LOGOUT, SYSOUT, etc. The AFTERDOxxx events run when continuing Lisp in the same incarnation following the SYSOUT, etc. (there is no such event for LOGOUT, of course). The ‘‘after’’ events are called in the same order in which the devices were defined; the ‘‘before’’ events in the reverse order.

For example, the BEFORELOGOUT event for the Leaf remote file server devices performs a FORCEOUTPUT on all its open files and then breaks the connection with the file server. The AFTERxxx events for the Leaf devices calls \REMOVEDEVICE on itself to flush any connection between the name and the server (since names and addresses can change over exit). The AFTERxxx events for the Dorado disk device rebuilds its cache of the disk’s directory.

There are a few devices in the system that exist only for their EVENTFN. In most cases, a simpler way to tell the system you want something performed around exit is to add your event function to the list AROUNDEXITFNS instead of going to the expense of defining a device for it. There is yet another list, \SYSTEMCACHEVARS, for handling a more specialized ‘‘around exit’’ operation: every time Lisp is booted, each of the variables in the list \SYSTEMCACHEVARS is set to NIL.

The following are required of all named devices, that is, devices that map from some hostname to the device, upon which files might be opened or otherwise manipulated:

(DIRECTORYNAMEP *HOST/DIR DEVICE*)

True if *HOST/DIR* is a valid directory name on *DEVICE*. Function should ideally perform recognition as well, and return the ‘‘true’’ name. For example, given ‘‘{PHYLEX:}<LISP>’’ as argument, it might return {Phylex:PARC:Xerox}<Lisp>. *HOST/DIR* might include a subdirectory name. The device should attempt to tell the truth about whether the subdirectory exists or not, though this may not be possible for devices with fake subdirectories. Defaults to NILL, i.e., device supports no directories. Used by the command CONN and the function DIRECTORYNAMEP.

(OPENFILE *NAME ACCESS RECOG PARAMETERS DEVICE*)

Used to implement the OPENFILE and OPENSTREAM functions. Opens the file named *NAME* on this device for access *ACCESS*, returning a STREAM. The stream is usually on *DEVICE* (its DEVICE field is *DEVICE*), but is not required to be. The arguments *ACCESS, RECOG, PARAMETERS* are as with the OPENFILE function in the manual. Thus, if *NAME* does not include a version number, recognition is according to *RECOG*, which should be appropriately defaulted per *ACCESS* (INPUT implies OLD, OUTPUT implies NEW, BOTH implies OLD/NEW).

The argument *NAME* can also be a STREAM, which must be a closed stream. OPENFILE should ‘‘reopen’’ the stream. The value returned in this case may be a new stream (with the same name as the old), or the old stream (*NAME*) itself. It is likely that the specification will be changed at some point to require that the old stream itself be returned, suitably reopened.

The argument *PARAMETERS* is a list of pairs (*OPTION VALUE*). The most interesting *OPTION*s are as follows:

`TYPE` — For new files, the type of the file (`TEXT` or `BINARY`). If this parameter is not specified, the value of the global variable `DEFAULTFILETYPE` (initially `TEXT`) should be used.

`CREATIONDATE` — For new files, the date of its creation. The device should use this if at all possible instead of letting the creation date default to the current date and time.

`LENGTH` — The intended length of the file, in bytes. This need not be accurate—it is only a hint that may allow smarter allocation. For example, if the device knows that it does not have room for a file of the specified length, it should immediately cause a `FILE SYSTEM RESOURCES EXCEEDED` error for the intended file.

`DON'T.CHANGE.DATE` — For old files being opened for access `BOTH`, don't change the creation date of the file. *ACCESS* = `BOTH` would normally imply that the content of the file is to change, and thus its creation date should be updated. Use of this parameter is a form of ‘‘cheating’’ to make it look as though the file had not changed. For example, the code that rewrites filemaps uses this parameter, since rewriting the filemap does not logically change the file's content.

`SEQUENTIAL` — If `T`, is a hint that the file will, or need, only be accessed sequentially, which may allow the device to open the file in a more efficient mode.

Any parameters that the device does not understand should be ignored, rather than be cause for an error. All devices are encouraged to support at least `TYPE` and `CREATIONDATE`.

The additional options `ENDOFSTREAMOP` and `BUFFERS` are handled by the generic file system code; specifying them is equivalent to calling `SETFILEINFO` (q.v.) immediately after the open.

Fine point about *ACCESS* = `OUTPUT`: this operation always produces a new, empty file, independent of whether its name is exactly the name of an existing file. That is, it replaces any old file by the same name. On opening, such a file has an end of file of zero. Of course, since *RECOG* defaults to `NEW` in this case, the name can only clash with an old file name if a version was explicitly specified, or *RECOG* is `OLD` or `OLD/NEW`. To open an old file for output but preserve its contents, i.e., only write over part of the file, one should open for *ACCESS* = `BOTH` (since to preserve the old contents one implicitly reads them).

Exception handling: If the desired file is not found, the `OPENFILE` method should return `NIL` rather than cause a `FILE NOT FOUND` error. This is so that the generic file system code can cause the error using the original file name, not the one packed with host and directory passed in to the `OPENFILE` method. The device should feel free to signal any other errors itself on failing to open the file, e.g., `FILE WON'T OPEN` for a busy file, `PROTECTION ERROR`, or `FILE SYSTEM RESOURCES EXCEEDED`. Ideally, this error should be signaled in a way that is resumable, i.e., so that a user could, in the break, take some action to remedy the condition and then type `OK` to continue. In most cases it suffices that all the internal functions below the `OPENFILE` be named with backslashes, so that the error code will choose to resume by reverting to the `OPENFILE` and trying again.

The device does not need to know about the set of open files (i.e., the value of `(OPENP)`), and in general should ignore it. That is, the device should perform the open as if there were no other files open and hence no conflict. The generic file system code looks at the stream returned from the OPENFILE method and then worries about whether there is actually another stream open by the same name. If there is, it closes the newly opened stream and then either returns the pre-existing stream, or causes a FILE WON'T OPEN error if the new and old access modes are in conflict. This design is crufty, but I believe it stems principally from the recognition problem—you don't know the full name of a file until you open it, so you can't tell until then whether you should have tried to open it in the first place. It will, of course, have to be completely changed when we go to multiple streams per file.

(REOPENFILE *NAME ACCESS RECOG PARAMETERS DEVICE OLDSTREAM*)

This is exactly like OPENFILE, except that it is called after LOGOUT (or other ''after'' events) on the name of any stream that was left open over exit. The idea is to maintain the illusion that the file really was open over LOGOUT, but check and make sure nothing changed. The generic file system code uses the VALIDATION field to test whether the file changed behind your back.

OLDSTREAM is the stream that was open before exit, and is supplied for the benefit of devices where there is no possibility that the file changed (e.g., {CORE}), so that they can just return *OLDSTREAM* directly. *OLDSTREAM* is also of use for those devices that have to cheat in order to maintain the illusion.

This will have to change when we go to multiple streams per file.

(GETFILENAME *NAME RECOG DEVICE*)

Performs ''recognition'' on *NAME*. That is, it returns the full name of the file that would be opened by OPENFILE in the indicated recognition mode, or NIL if the file is not found. It is not necessary that OPENFILE actually be capable of opening the file (there is no need to check protection, for example). Used by INFILEP, OUTFILEP, FULLNAME.

(DELETEFILE *NAME DEVICE*)

Deletes the file named *NAME*, returning its full name on success, NIL on failure. Recognition mode is implicitly OLDEST. Local devices, after recognizing the file, should make sure that it is not OPENP (open files can not be deleted). This and RENAMEFILE are usually the only device methods that need to know anything about what files are open.

(GENERATEFILES *DEVICE PATTERN DESIREDPROPS OPTIONS*)

Enumerates files matching *PATTERN*. Returns a ''file generator object'' of the form (*NEXTFILEFN INFOFN . ArbitraryState*). This is described in more gory detail under **Directory Enumeration**.

(RENAMEFILE *OLDNAME NEWNAME DEVICE*)

Renames the file named *OLDNAME* to have name *NEWNAME*. Returns the full name of the new file if successful, NIL if not. Recognition mode is implicitly OLD for *OLDNAME*, NEW for *NEWNAME*. The generic file system code invokes this method to implement the function RENAMEFILE only when the host fields of both filenames map to the same device. Defaults to \GENERIC.RENAMEFILE, which is also the function that the system calls when the old and new names are on different devices. \GENERIC.RENAMEFILE is defined to copy *OLDNAME* to *NEWNAME* and then delete *OLDNAME*.

The following methods are invoked for open streams. They are all required:

(BIN *STREAM*)

Returns the next byte of input from *STREAM*, or takes the appropriate action if at end of file. Unless a device has a good reason not to, it should call (\EOF.ACTION *STREAM*) at end of file/stream.

The device `BIN` method is actually not used directly. Rather, every stream has a `STRMBINFN` field, which is the function actually applied to do the input. The `STRMBINFN` field could thus be used to fake a specialization of the device differing only in the `BIN` method. However, the typical use of `STRMBINFN` is to temporarily override the device default. In particular, setting a stream's access to `INPUT` or `BOTH` automatically sets the stream's `STRMBINFN` to be the device's `BIN` method; setting access to `NIL` or `OUTPUT` sets the `STRMBINFN` to be an error. This relieves the device's `BIN` method of any need to check the stream's access on every call to `BIN`. Some network streams temporarily set their `STRMBINFN` to be an input eater when they receive a ‘‘clear output’’ command.

Currently, all Interlisp-D streams have bytesize 8, so `BIN` always returns an 8-bit integer.

Calls to the function `BIN` are compiled into the `BIN` opcode, which runs in microcode on some machines if the requirements for it are met. More on this later.

(`BOUT` *STREAM BYTE*)

Outputs *BYTE* to *STREAM*. As with `BIN`, this method is not used directly. Rather, every stream has a `STRMBOUTFN` field, which is the function actually applied to do the output. Setting a stream's access to `OUTPUT` or `BOTH` automatically sets the stream's `STRMBOUTFN` to be the device's `BOUT` method.

There exists a `BOUT` opcode, but the design is incomplete.

(`PEEKBIN` *STREAM NOERRORFLG*)

Returns the next input byte from *STREAM*, but does not advance the stream pointer. Thus a subsequent `PEEKBIN` or `BIN` will return the same byte. At end of stream, the device should take eof action as with `BIN`, unless *NOERRORFLG* is true, in which case it should return `NIL`.

(`READP` *STREAM FLG*)

Returns true if input is available from *STREAM*, that is, if a `BIN` right now would succeed without waiting. Defaults to `\GENERIC.READP`, which uses `EOFP` and `PEEKBIN`.

Roughly speaking, `READP` is the complement of `EOFP` for streams that are not arriving in real time. It is interestingly different for network streams, or the keyboard.

FLG is a bit of cruft that not everyone pays attention to, and may be flushed at some point: if *FLG* is `NIL`, then `READP` should return `NIL` if the only input waiting is an end of line character.

(`EOFP` *STREAM*)

Returns true when *STREAM* is ‘‘at end of file’’, i.e., a `BIN` would cause an end of file action to occur. Note that for a network stream, it is possible for both `EOFP` and `READP` to be false simultaneously, viz. when there is no input waiting (buffered locally), but the remote end of the stream has not indicated that there is no more input.

There are some who call `EOFP` on streams open only for output. This is a crock; output streams are always at end of file. But to avoid complaints, a device could return `T` for `EOFP` on an output stream.

(`BLOCKIN` *STREAM BUFFER BYTEOFFSET NBYTES*)

Performs bulk input transfer: retrieves the next *NBYTES* bytes from *STREAM* and stores them in successive byte positions in *BUFFER* starting at *BYTEOFFSET*. Defaults to `\GENERIC.BINS`, which repeatedly calls `BIN` and `\PUTBASEBYTE`.

It is almost always the case that a device with a non-trivial `BLOCKIN` method can be made to be a Buffered device, thereby benefiting from other Buffered operations as well.

(`BLOCKOUT` *STREAM BUFFER BYTEOFFSET NBYTES*)

Performs bulk output transfer: outputs *NBYTES* bytes to *STREAM*, taking the bytes from *BUFFER* starting at *BYTEOFFSET*. Defaults to `\GENERIC.BOUTS`, which repeatedly calls `\GETBASEBYTE` and `BOUT`.

(`FORCEOUTPUT` *STREAM WAITFORFINISH*)

Forces to its ultimate destination any output buffered on *STREAM* but not yet sent. *WAITFORFINISH* means that the function should not return until it is confident that the output has reached its destination and been committed. Defaults to `NILL`, which is reasonable for unbuffered streams.

For example, for a network stream, `FORCEOUTPUT` sends the current packet being buffered up. For a buffered stream to the disk, `FORCEOUTPUT` writes out to the disk any ‘‘dirty’’ pages, and makes sure the file is in such a state that if the machine were booted after `FORCEOUTPUT` returns, that the file could be successfully reopened with no information lost.

(`GETFILEINFO` *NAME/STREAM ATTRIBUTE DEVICE*)

Returns the value of the specified *ATTRIBUTE* of *NAME/STREAM*, which can be an open Stream or the name of a (closed) file. Returns `NIL` for attributes it doesn’t know about. It is considered good citizenship, though not absolutely required, to know about the following attributes:

`LENGTH` Length of the stream/file in bytes. If the device’s method returns `NIL`, but the stream is random access, the generic `GETFILEINFO` code tries the device’s `GETEOFPTR` method instead.

`SIZE` Length in pages, i.e., `(FOLDHI` length `BYTESPERPAGE)`.

`CREATIONDATE` Date when the file’s contents were created, as a string. The creationdate does not change when a file is copied or renamed, only when it is changed.

`WRITEDATE` Date when the file was written to its current place of storage.

`READDATE` Date when the file was last read.

`ICREATIONDATE`, `IWRITEDATE`, `IREADDATE`
The creation, write and read dates as integers, such as from the function `IDATE`.

`TYPE` Type of the contents: `TEXT` for files that contain only ‘‘text’’ (generally meaning 7-bit ascii), `BINARY` for all others. `NIL` means unknown.

`AUTHOR` Name of the user who created the file (a string).

The following ‘‘generic’’ attributes are generally handled by the generic side of `GETFILEINFO` if the device’s `GETFILEINFO` method returns `NIL`:

`EOL` The end of line convention of the stream (`CR`, `CRLF` or `LF`).

`BUFFERS` The number of pagemap buffers for use by the stream (see description of `MAXBUFFERS` field of pagemapped streams).

`ENDOFSTREAMOP` Action to take on any attempt to read beyond the end of file. This is a function of one argument, the stream. The function can cause an error, or return a value, which is interpreted as a value to return from `BIN`. The default `ENDOFSTREAMOP` causes an `END OF FILE` error.

`ACCESS` An atom describing the access mode of the stream (`INPUT`, `OUTPUT`, etc). This is so generic that it is handled before the device's method ever sees it.

`BYTESIZE, OPENBYTESIZE` The size of bytes transmitted on the stream. Always 8 these days.

(`SETFILEINFO` *NAME/STREAM ATTRIBUTE VALUE DEVICE*)

Sets the value of the specified *ATTRIBUTE* of *NAME/STREAM* to be *VALUE*. Returns `T` if successful, `NIL` if unsuccessful, or for attributes it doesn't know about.

It is not generally required that `SETFILEINFO` recognize any attributes at all—`NILL` is a perfectly good filler for this slot. Most devices recognize no more than `TYPE` and `CREATIONDATE` (`ICREATIONDATE`), and even those are not very important, as most applications set those attributes in the *PARAMETERS* argument to `OPENFILE` when creating a file.

ATTRIBUTE = `LENGTH` implies actually truncating (or lengthening) the file; however, the `SETFILEINFO` need not handle this itself—if it returns `NIL`, then the generic file system will attempt to use the `SETEOFPTR` method instead.

The following operations are only required of random access streams. They default to the function `\IS.NOT.RANDACCESSP`, which causes a ''Stream is not randaccessp'' error when called.

(`GETFILEPTR` *STREAM*)

Returns the current file pointer (byte position) in *STREAM*. The file pointer is zero when the stream is opened (except for *ACCESS* = `APPEND`), and is incremented by one for each byte read.

Although this operation is only absolutely required for random access streams, it is desirable to supply it for other streams where possible. For example, when reading a file sequentially through PupFtp, the stream can count the bytes as they go by and thus give an accurate value for `GETFILEPTR`. If a stream has no idea at all of position, it can make its `GETFILEPTR` be the function `ZERO` and thereby at least avoid breaks from code that calls `GETFILEPTR` carelessly.

(`GETEOFPTR` *STREAM*)

Returns the file pointer of the end of *STREAM*, i.e., the file pointer that `GETFILEPTR` would return after the last byte of *STREAM* is read. Same as the `LENGTH` attribute for a stream that represents a file. Of course, non-random access streams may have no idea where the end is, and causing a non-randaccessp error is perfectly acceptable.

(`SETFILEPTR` *STREAM BYTENUMBER*)

Sets the file pointer of *STREAM* to be *BYTENUMBER*. The special value *BYTENUMBER* = ÿ1 means the end of the stream; other negative values are illegal.

`SETFILEPTR` beyond the end of the stream is permissible, but it has no immediate effect beyond changing the logical file pointer. Attempting to then `BIN` causes an `EOF` error. Attempting to `BOUT` (for a file open for write) should extend the file, so that its eof is immediately beyond the newly `BOUT`ed byte.

As with `GETFILEPTR`, there is no requirement that this work on non-random access streams, and it may be completely impossible on some of them. However, for those non-random access streams that perform `GETFILEPTR`, it is possible to fake `SETFILEPTR` for values larger than the current file pointer by skipping some number of bytes in the file, e.g., by performing `(RPTQ (DIFFERENCE` *BYTENUMBER* `(GETFILEPTR` *STREAM*`)) (BIN` *STREAM*`))`. There are some applications for which forward `SETFILEPTR` is all the random access that is actually required, so it is nice to be able to accommodate such applications.

`(BACKFILEPTR` *STREAM*`)`

Backs up the file pointer in *STREAM* by one byte. Functionally the same as `(SETFILEPTR` *STREAM* `(SUB1 (GETFILEPTR` *STREAM*`))`, but may be possible on non-random access streams by maintaining a one-character buffer, which is all the backing up this operation is formally required to perform. I believe the main use for this is in `READ`, which needs to back up the stream one character when, for example, it reads a break character terminating an atom.

`(SETEOFPTR` *STREAM* *LENGTH*`)`

Changes the length of *STREAM* to be *LENGTH*, i.e., ‘‘sets’’ its end of file pointer. This may require lengthening or truncating the file. Used by the function `\SETEOFPTR` and by `SETFILEINFO` for attribute `LENGTH` when the device’s `SETFILEINFO` method doesn’t handle it.

The following three fields are place holders for possible future extensions. These fields are not currently used at all:

`(LASTC` *STREAM*`)`

Returns the last character read from *STREAM*, i.e., the last byte that was `BIN`ed, as a character. `LASTC` is currently implemented via `BACKFILEPTR`.

`(FREEPAGECOUNT` *HOST/DIR* *DEVICE*`)`

Intended use is to return the number of free pages on *HOST/DIR*. May be folded into a general GET/SET device/directory info operation.

`(MAKEDIRECTORY` *HOST/DIR* *DEVICE*`)`

Intended use is to create a new directory *HOST/DIR*.

The remaining fields in the `FDEV` are for buffered and page-mapped streams, and are ignored for non-buffered devices. These fields are described in separate sections.

Streams

The following fields are used by all streams:

`DEVICE`	Pointer to this stream’s `FDEV`.
`FULLFILENAME`	‘‘Full’’ name by which this file is known to the user. Should be an uppercase litatom, fully qualified so that giving the same name back to the file system should produce the same file (to the extent that the device can support such uniqueness). Is `NIL` for unnamed streams.
`FULLNAME`	Access field. Is the same as `FULLFILENAME`, unless that is `NIL`, in which case it is the stream itself. This avoids the circularity that would result if the FULLFILENAME field contained the stream datum.

NAMEDP	Access field. Is T if the streams is named, i.e., its FULLFILENAME is non-NIL.
ACCESSBITS	Contains a numeric code describing what access mode the file is open for: there are read, write and append bits. This field is usually accessed indirectly via the ACCESS field. However, there are macros for referring to particular types of access using more efficient bit test operations:

(OPENED *STREAM*)	ACCESS is not NIL.
(READABLE *STREAM*)	Read bit is on: ACCESS is INPUT or BOTH.
(READONLY *STREAM*)	Only the read bit is on: ACCESS is INPUT.
(APPENDABLE *STREAM*)	Append bit is on: ACCESS is OUTPUT, BOTH or APPEND.
(APPENDONLY *STREAM*)	Only the append bit is on: ACCESS is APPEND.
(DIRTYABLE *STREAM*)	Append or write bit is on: ACCESS is OUTPUT, BOTH or APPEND. Yes, this is operationally the same as APPENDABLE, given the four possible values of ACCESS.
(OVERWRITEABLE *STREAM*)	Write bit is on: ACCESS is OUTPUT or BOTH.
(WRITEABLE *STREAM*)	Write bit is on, or append bit is on and file is at EOF. Avoid using this one, it's a little strange.

ACCESS	Access field for referring to the ACCESSBITS field symbolically. Its value is one of the legal values of the *ACCESS* argument to OPENFILE: INPUT, OUTPUT, BOTH, APPEND; or NIL when the stream is closed. Replacing this field has the side effect of setting the BINABLE, BOUTABLE, STRMBINFN and STRMBOUTFN fields appropriately (from the corresponding device fields, or to values consistent with no access).
USERCLOSEABLE	Flag, true if the stream can be closed by CLOSEF. Default is T, but is NIL for such things as dribble files and the terminal.
USERVISIBLE	Flag, true if the stream is to be listed in the result of (OPENP). Default is T, but is NIL for such things as dribble files and the terminal.
BINABLE	True if BIN microcode can be used. Normally set automatically from FDBINABLE when input access is set.
BOUTABLE	True if BOUT microcode can be used. Normally set automatically from FDBOUTABLE when output access is set.
EXTENDABLE	True if BOUT can extend the buffer when COFFSET reaches CBUFSIZE. Obsolete.
STRMBINFN	Function called by BIN. This is normally set indirectly as a side effect of setting the ACCESS field. Setting ACCESS to an input access (INPUT or BOTH) sets the STRMBINFN to be the stream's device's BIN method. Setting to any other access sets the STRMBINFN to be a ‘‘file not open’’ trap.
STRMBOUTFN	Function called by BOUT. As with STRMBINFN, this is normally set indirectly (from the device's BOUT method) as a side effect of setting the ACCESS field.

OUTCHARFN — Function called to output a single byte. This is like STRMBOUTFN, except for being one level higher: it is intended for text output. Hence, this function should convert (CHARCODE EOL) into the stream' actual end of line sequence, and should adjust CHARPOSITION appropriately before invoking the stream's STRMBOUTFN to actually put the character. Defaults to \FILEOUTCHARFN. The OUTCHARFN for the display additionally worries about such things as ECHOCONTROL.

CHARPOSITION Current horizontal character position in the stream. Incremented (and reset to zero) by OUTCHARFN. Used by the function POSITION.

LINELENGTH Maximum line length of the stream, in characters. Used by the function LINELENGTH. Defaults (at creation time) to the value of the global variable FILELINELENGTH.

EOLCONVENTION The stream's end of line convention: the manner in which ‘‘end of line’’ is encoded on this stream. That is, output of an end of line (function TERPRI) produces the stream's end of line sequence, and on input, the stream's end of line sequence is converted to (CHARCODE EOL) by READC. This is not necessarily the same as the way that end of line is encoded in the actual file written by, say, a file server. For example, Lisp might open a stream to a Tenex file server with EOLCONVENTION of CR, while the server might choose to take each of the CRs in the stream and actually store a CR, LF sequence in the physical file.

The convention is encoded as a two-bit field; the constants CR.EOLC, LF.EOLC, CRLF.EOLC can be used to refer to the currently known values symbolically. Default in Interlisp-D is CR.EOLC.

ENDOFSTREAMOP Function of one argument (the stream) called when an attempt to read beyond the end of file occurs. If this function returns something, it should be interpreted as a value to return from BIN (the value T is currently prohibited). Defaults to \EOSERROR, which causes an END OF FILE error.

VALIDATION Pointer field, some compact encoding of the state of the file such that if the file's content changes, the VALIDATION changes. The file's ICREATIEONDATE attribute usually works well enough. The only use for this field is to check whether the file changed over LOGOUT, etc.—if the VALIDATION of the stream returned from REOPENFILE is EQUAL to the VALIDATION of the stream open before LOGOUT, the stream is assumed to be unchnaged. This will probably be the sole concern of the device when we go to multiple streams per file.

BYTESIZE — Byte size of the file, i.e., what BIN and BOUT traffic in. Defaults to 8. This field is not used by many; there are probably a lot of things that won't work if the byte size is not 8.

OTHERPROPS — List in property list format used by the function STREAMPROP. Analogous to WINDOWPROP, etc.

IMAGEOPS — Image operations vector (object of type IMAGEOPS) for use of device-independent graphics operations, such as DSPXPOSITION, DSPFONT. Defaults to \NOIMAGEOPS, a vector suitably defined for non-display devices. See the implementors' manual chapter **Device-Independent Graphics**.

IMAGEDATA — Device-dependent data for use by IMAGEOPS.

REVALIDATEFLG — Flag. The standard use of this flag is to solve a problem with correctly maintaining the creation date. The problem is that the definition of ‘‘creation date’’ is that the creation date changes whenever the contents of the file change. If followed literally, this would mean, for example, that ever time you wrote out a page of a {DSK} file, you would also have to rewrite its leader page with a new creation date. However, it suffices in practice to only change the creation date when it would matter, i.e., when there would be any possibility of some agent other than the currently running Lisp to see the change. Usually, this means the only time to worry about is when the Lisp vmem is saved and a file that was open before the save is written to again afterwards.

Thus, the use of this flag (for those devices that care) is as follows: the device's `BEFORExxx` events set this flag true for any streams open on the device. Then, whenever the device is about to do something that would change the file's content, e.g., write out a new page, it first tests `REVALIDATEFLG`. If the flag is true, it updates the file's creation date and clears the flag.

`NONDEFAULTDATEFLG` Flag. Standard use is in conjunction with `REVALIDATEFLG`, to mark a file that was opened in a way that the user constrained the creation date of the file (e.g., the *PARAMETERS* argument to `OPENFILE` included an explicit creation date, or the option `DON'T.CHANGE.DATE`).

`F1`, `F2`, `F3`, `F4`, `F5` Pointer fields for private use by the stream, to maintain stream-specific state of concern only to the device. Stream clients that wish to hang information on a stream without regard to what kind of stream it is should use the function `STREAMPROP`.

`FW6`, `FW7`, `FW8`, `FW9` 16-bit word fields for private use by the stream.

`DIRTYBITS` Obsolete.

`EXTRASTREAMOP` ?

Buffered Streams

Buffered streams are ones that constrain themselves to obey a set of conventions that make it easy for an agent (e.g., microcode) to perform input or output on the stream without knowing about the details of the stream's physical i/o. The stream maintains a ‘‘current buffer’’ and two indices into that buffer, the offset of the next byte, and the offset of the end of the buffer. As long as the former index is less than the latter, the stream guarantees that the bytes in the buffer between those indices are the true contents of the file/stream starting at the current file pointer. Advancing the first index effectively advances the file pointer. When it reaches the second index, a stream-specific operation is called to ‘‘refill’’ the buffer.

The following fields are used by buffered streams:

`COFFSET` Byte offset in the buffer `CBUFPTR` of the next `BIN` or `BOUT`.

`CBUFSIZE` ‘‘Size’’ of the current buffer, i.e., byte offset that is one beyond the last byte.

`CBUFMAXSIZE` For output, the maximum size the buffer can be written to. If `COFFSET` reaches `CBUFSIZE`, but `CBUFSIZE` is less than `CBUFMAXSIZE`, then the buffer can be extended.

`CBUFPTR` Pointer to current buffer. Must be valid if `COFFSET` is less than `CBUFSIZE` and `BINABLE` or `BOUTABLE` is true. It is not necessary that this ‘‘buffer’’ be anything other than some chunk of memory, a portion of which contains interesting data. Thus, the bytes from offset `COFFSET` to `CBUFSIZE` must be valid, but `COFFSET` need not start at zero, nor need `CBUFSIZE` or `CBUFMAXSIZE` coincide with the end of the underlying structure.

`CBUFDIRTY` Flag, true if current buffer has been written to.

In general, the device has sole responsibility for setting `CBUFSIZE`, `CBUFMAXSIZE`, and `CBUFPTR`; generic code does not touch those. The fields `COFFSET` and `CBUFDIRTY` can be changed by generic stream clients as well as by device-specific code. For example, code that simulates a `BIN` increments `COFFSET`; code that writes directly to the stream's buffer sets `CBUFDIRTY` true.

The following methods are defined for devices implementing buffered streams:

(`GETNEXTBUFFER` *STREAM WHATFOR NOERRORFLG*) [Device method]

> Called when *STREAM* needs to have its buffer fixed, i.e., the state of *STREAM* is such that `BIN` (*WHATFOR* = `READ`) or `BOUT` (*WHATFOR* = `WRITE`) cannot proceed. This method should do whatever is necessary to allow the operation to proceed. This typically includes disposing of the current buffer somehow (if `GETNEXTBUFFER` was invoked because the buffer was exhausted), and fetching a new buffer consistent with *STREAM*'s current position.
>
> In the case of *WHATFOR* = `READ`, `GETNEXTBUFFER` returns `T` on success, i.e., if *STREAM* is not at end of file. When *STREAM* is at end of file, `GETNEXTBUFFER` should take standard end of stream action, returning whatever `\EOF.ACTION` returns (if anything). However, if *NOERRORFLG* is true, `GETNEXTBUFFER` should just return `NIL` immediately.

(`RELEASEBUFFER` *STREAM BUFFER*) [Device method]

> Performs any device-specific operation required when *BUFFER*, which is the current value of *STREAM*'s `CBUFPTR` field, is ‘‘released’’ (when the `CBUFPTR` field is replaced). This is used so that different pagemap-like devices can share certain code. For example, in the case of pagemapped streams, `RELEASEBUFFER` marks the buffer dirty in the case that the stream's `CBUFDIRTY` field has been set.
>
> This method is not currently used.

The functions `\BUFFERED.BIN`, `\BUFFERED.PEEKBIN`, `\BUFFERED.BOUT`, `\BUFFERED.BINS` and `\BUFFERED.BOUTS` are supplied for use by buffered streams; they are standardly used to implement the `BIN`, `PEEKBIN`, `BOUT`, `BLOCKIN` and `BLOCKOUT` device methods. In addition, the function `COPYBYTES`, when presented with a source stream that is buffered, utilizes the `GETNEXTBUFFER` method to efficiently copy bytes to the destination a buffer-full at a time.

Pagemapped Streams

Pagemapped streams are a particular kind of random access Buffered stream that buffers its data in units of pages. The device provides methods that read or write data in units of pages, while system-supplied Pagemapped functions handle the responsibilities of a Buffered stream, as well as managing the file pointer for random access. In general, a stream can have several pages of a file buffered at a time, allowing the code to make some effort to make efficient use of multi-paged transfers where applicable.

To create a pagemapped device, create an `FDEV`, fill in the necessary private fields, then call the following function:

(`\MAKE.PMAP.DEVICE` *DEVICE*) [Function]

> Fills in fields in the device appropriate for pagemapped devices, and returns the updated device. The fields it fills are the flag fields `FDBINABLE`, `FDBOUTABLE`, `RESETABLE`, `RANDOMACCESSP`, `PAGEMAPPED`, `BUFFERED` (all true), and the methods `BIN`, `BOUT`, `PEEKBIN`, `BLOCKIN`, `BLOCKOUT`,

READP, EOFP, GETFILEPTR, BACKFILEPTR, SETFILEPTR, GETEOFPTR, SETEOFPTR, GETNEXTBUFFER and FORCEOUTPUT.

A Pagemapped device is required to supply the following methods (in addition to those required of all devices and not filled in by \MAKE.PMAP.DEVICE):

(READPAGES *STREAM FIRSTPAGE# BUFFERS*) [Device method]

Causes pages of *STREAM* to be read into *BUFFERS*. The first page read is *FIRSTPAGE#* (zero for the first page of the file). *BUFFERS* is either a single page-sized buffer (a VMEMPAGEP), in which case exactly one page is read, or it is a list of such buffers. READPAGES returns the total number of bytes read. If the last page read is not a full page, READPAGES should zero out the rest of its buffer. READPAGES can assume that the buffers are page-aligned, although they need not be consecutive.

(WRITEPAGES *STREAM FIRSTPAGE# BUFFERS*) [Device method]

Writes data from *BUFFERS* out to *STREAM*. The first page written is *FIRSTPAGE#*. *BUFFERS* is as with READPAGES.

Neither READPAGES nor WRITEPAGES affects *STREAM*'s file pointer or end of file; those are managed by higher-level pagemapped routines. WRITEPAGES might, however, want to look at *STREAM*'s EPAGE and EOFFSET fields if it needs to take any special action around the end of the file. It is possible, for no particularly good reason, for READPAGES to get called for a page beyond the end of file; in fact, this standardly happens when writing a new file. The READPAGES method in this case should just clear the buffer and return zero.

(TRUNCATEFILE *STREAM PAGE# OFFSET*) [Device method]

Truncates *STREAM* so that its end of file is *PAGE#*, *OFFSET*, which should be defaulted to *STREAM*'s EPAGE and EOFFSET. Can be used to either shorten or lengthen a file; if lengthening, the file should be padded with nulls. Used by \PAGED.SETEOFPTR and \PAGED.FORCEOUTPUT. As of this writing there are still bugs in this code in certain funny cases, such as when you SETFILEPTR beyond eof and then BOUT.

The following fields of a stream are meaningful for a pagemapped device. The generic pagemapped codes maintain them as operations on the file are performed, but they should all be initialized appropriately by the device's OPENFILE method:

CPAGE — For pagemapped streams, the current page position in the stream. Together with COFFSET, this constitutes the stream's file pointer. The device's OPENFILE method should set CPAGE and COFFSET to zero, except for files opened with access APPEND, in which case they should be set to the end of file.

EPAGE, EOFFSET — For pagemapped files, the page and byte offset of the end of file. Note that this is the *logical* end of the file; it need have nothing to do with the physical end of file, except that when a file is closed, the device should see to it that its logical and physical EOFs are the same (normally seen to by the TRUNCATEFILE inside of \CLEARMAP, below). In fact, as a typical file is being written, EPAGE tends to stay several pages ahead of the physical end of file by virtue of the fact that pages are being buffered before being written out.

BUFFS — For pagemapped streams, a pointer to the stream's BUFFER chain. Initially NIL (no buffers allocated). The device usually has no direct interest in this field.

MAXBUFFERS — For pagemapped streams, the maximum number of buffers desired in the stream's BUFFS chain. If the code needs another buffer and there are already MAXBUFFERS buffers, it will try to recycle the least recently referenced buffer. Defaults to `\STREAM.DEFAULT.MAXBUFFERS`. The user can change this field for an open stream by calling `SETFILEINFO` with attribute `BUFFERS`.

`MULTIBUFFERHINT` Flag. For pagemapped streams, is a hint to the pagemap code that the device prefers to transfer data more than one buffer at a time. If this flag is true, the pagemap code tries to write out (`WRITEPAGES`) more than one buffer at a time when the opportunity arises. A similar improvement is planned, but not implemented, for reading multiple buffers at a time.

The following functions are of use for pagemapped devices:

(`\PAGED.FORCEOUTPUT` *STREAM WAITFORFINISH*) [Function]

This function implements the `FORCEOUTPUT` method for pagemapped streams: it causes any dirty pages to be written out (using `WRITEPAGES`), then calls the `TRUNCATEFILE` method to set the end of file.

This function is normally installed as the `FORCEOUTPUT` method by the function `\MAKE.PMAP.DEVICE`. However, the device can override this default (by supplying its own function in that field), in which case it might want to call the function `\PAGED.FORCEOUTPUT` explicitly as part of its more comprehensive `FORCEOUTPUT` method.

There is an unpleasantness in the implementation of pagemapped devices that stems from the fact that originally all devices (the few that existed in the distant past) were made to support the `PMAP` package, a means whereby a programmer could get direct access to the buffers of a file, much as one can with the `PMAP` JSYS in Tenex. As a result, the buffers used by pagemapped streams are set up in a special manner so that the garbage collector can tell when the user no longer has access to a `PMAP` buffer. The PMAP package is being phased out.

This is all exceedingly crufty, and is of little concern to the device implementer, except for the fact that it requires that the buffers be explicitly released when a stream is closed; the buffers are not automatically collected when the stream is dropped.

(`FORGETPAGES` *STREAM FROMPAGE TOPAGE*) [Function]

‘‘Forgets’’ pages *FROMPAGE* thru *TOPAGE* of *STREAM*; i.e., removes those pages from the set of pages being currently buffered, and frees the buffers they were occupying. If *FROMPAGE* = *TOPAGE* = `NIL`, forgets all pages, and releases all of *STREAM*’s buffers.

(`\CLEARMAP` *STREAM*) [Function]

Performs a `FORCEOUTPUT` (if *STREAM* is open for output) followed by a `FORGETPAGES`. This is the standard action that should be taken by a pagemapped stream’s `CLOSEFILE` method.

Directory Enumeration

This section describes how directory enumeration works—what you need to know in order to implement the `GENERATEFILES` device method, and what you need to know as a programmer trying to enumerate a directory via anything more elaborate than the function `DIRECTORY`.

The general idea is that the directory enumeration code is given a pattern, and it returns a generator that, each time it is poked, returns another file name matching the pattern. In addition, the generator provides a handle for getting file attributes of each enumerated file. This second handle is important for efficiency: although one could just take the file name given by the enumerator and pass it to `GETFILEINFO`, the device, in the course of enumeration, usually has its fingers on the file closely enough that it need not perform the second directory lookup that a `GETFILEINFO` out of the blue would require. The caller of the directory enumeration code specifies ahead of time which, if any, attributes will be required (a necessity for most file server implementations).

Information for device implementors. A *file generator* is an object represented as a list described by the record `FILEGENOBJ`, exported from `FILEIO`:

```
(RECORD FILEGENOBJ (NEXTFILEFN FILEINFOFN . GENFILESTATE))
```

`NEXTFILEFN` and `FILEINFOFN` are functions of the device's choosing that when called will return the next file, and attributes for that file. `GENFILESTATE` is arbitrary state maintained by the generator. With that as background, here are the pieces of directory enumeration:

(`GENERATEFILES` *DEVICE PATTERN DESIREDPROPS OPTIONS*) [Device method]

Returns a generator that enumerates files matching *PATTERN*, which is a string that has host and directories suitably filled in from defaults, and may contain the pattern character ''*'' to match an arbitrary number of characters. *DESIREDPROPS* is a list of file attributes that may be requested during the enumeration; they must be valid *ATTRIBUTE* arguments to `GETFILEINFO`. *OPTIONS* is a list of options to the enumeration, chosen from among the following:

`SORT` The files should be enumerated in sorted order. If this option is not specified, the device is free to enumerate files in any convenient order.

There is some question as to whether files should be enumerated lowest version first (as IFS's do) or highest version first (as Twenex does). I prefer the latter, but given servers that do the former, we currently make no requirement about version order.

`RESETLST` Informs the enumerator that the enumeration context is surrounded by a `RESETLST`, so that it may perform `RESETSAVE`s to clean up after itself if the enumeration is aborted. Cleaning up can be a very messy business without this information about the scope of the enumeration, so all callers of `\GENERATEFILES` are strongly encouraged to provide it.

`GENERATEFILES` should return a file generator with a suitable `NEXTFILEFN` and `FILEINFOFN`.

Fine point about missing fields in the pattern: null fields in *PATTERN* match only files for which the corresponding field is null. A null version is interpreted as highest. Thus,

`DIR *` = `DIR *.*` = `DIR *.;*` enumerates everything.

`DIR *.` = `DIR *.;*` enumerates all versions of files with null extension.

`DIR *.;` enumerates highest version of files with null extension.

`DIR *.*;` enumerates highest version of everything.

It is difficult for some devices to enumerate only highest version of files; there are several devices in the system that treat a null version the same as version `*`. However, every device should try its best. With some work, any device that can enumerate all versions can enumerate just highest version if it enumerates in sorted order and uses perhaps a little lookahead to assure that any name it returns is the one of highest version.

(`NEXTFILEFN` *GENFILESTATE NAMEONLY*) [File Generator Component]

Generates the next file, returning its name as a string, or `NIL` if the generator is exhausted. *GENFILESTATE* is the state component of the file generator returned from `GENERATEFILES`. *NAMEONLY* means that the caller is only interested in the file's `Name.Ext` fields, not the full file name (and no more than one version of the file need be enumerated); however, it is always permissible to return the full file name. The *NAMEONLY* option is used by `SPELLFILE`.

`(FILEINFOFN` *GENFILESTATE ATTRIBUTE*`)` [File Generator Component]

Returns the value of the *ATTRIBUTE* property of the file most recently generated by the `NEXTFILEFN`, i.e., effectively `(GETFILEINFO` latest-name *ATTRIBUTE*`)`, but hopefully much faster. *ATTRIBUTE* must have been a member of the *DESIREDPROPS* argument to `GENERATEFILES`.

Not all device implementors are enthused about implementing a pattern matcher for file names. The following functions are provided to help out:

`(DIRECTORY.MATCH.SETUP` *PATTERN*`)` [Function]

Accepts as *PATTERN* a file name string such as passed to `GENERATEFILES`. Returns an object suitable as a filter to `DIRECTORY.MATCH`.

`(DIRECTORY.MATCH` *FILTER TESTNAME*`)` [Function]

Matches *TESTNAME*, a file name, against *FILTER*, the object returned from `DIRECTORY.MATCH.SETUP`. Returns true if *TESTNAME* matches the pattern, false if not. The match is case-insensitive.

`(\NULLFILEGENERATOR)` [Function]

Returns a file generator that produces no files.

`(\GENERATENOFILES` *DEVICE PATTERN DESIREDPROPS OPTIONS*`)` [Function]

Returns a ''stupid'' file generator for devices that don't know how to enumerate in general. If *PATTERN* contains no wildcards, but names a file that is `INFILEP`, then the generator produces exactly that file. If *PATTERN* contains a wildcard in the version field, it uses `GETFILENAME` to laboriously generate all the versions of the file. In all other cases, `\GENERATENOFILES` returns a null file generator.

Information for clients of device enumeration. The following functions make up the ''public'' interface to directory enumeration:

`(\GENERATEFILES` *PATTERN DESIREDPROPS OPTIONS*`)` [Function]

Returns a file generator object for enumerating the files matching *PATTERN*. *PATTERN* is expanded by adding the default host and/or directory if appropriate. See description of the `GENERATEFILES` method for description of *DESIREDPROPS* and *OPTIONS*.

`(\GENERATENEXTFILE` *GENERATOR NAMEONLY*`)` [Function]

Returns the next file, as a string. *GENERATOR* is the object returned from `\GENERATEFILES`; *NAMEONLY* indicates caller does not require that the full name be returned, but that the name and extension are sufficient.

`(\GENERATEFILEINFO` *GENERATOR ATTRIBUTE*`)` [Function]

Returns the value of the *ATTRIBUTE* property of the file most recently generated by \GENERATENEXTFILE, i.e., effectively (GETFILEINFO latest-name *ATTRIBUTE*). *ATTRIBUTE* must have been a member of the *DESIREDPROPS* argument to \GENERATEFILES.

(DIRECTORY.FILL.PATTERN *PATTERN DEFAULTEXT DEFAULTVERS*) [Function]

This function is used to fill in defaults in *PATTERN* before passing it to \GENERATEFILES. If *PATTERN* does not include an extension or version, but those fields are not explicitly omitted (e.g., ‘‘FOO’’, but not ‘‘FOO.’’; ‘‘FOO.BAR’’, but not ‘‘FOO.BAR;’’), they are filled in with *DEFAULTEXT* and *DEFAULTVERS*, which themselves default to ‘‘*’’. This function is used by the DIR command, and should probably be used by any code that takes a user-supplied pattern and enumerates files from it.

"TEDIT BEHIND EXEC WINDOW"
PRELIMINARY DESIGN NOTES
JIM BLUM
6/3/85

This paper is a list of issues and alternatives to designing and implementing TEDIT behind EXEC windows.

KNOWN ISSUES TO BE SOLVED

1. Certain functions which normally expect a display stream as the argument have to be changed to accept a textstream as a valid type of stream (or whatever kind of stream it ends up being, for now I will refer to it as a TEDITSTREAM). Some example functions are TTYDISPLAYSTREAM, (or whatever mechanism we use to replace WFROMDS functionality).

2. There has to be a place holder or mechanism by which it is know to TEDIT where the already processed text ends, and the current edible text starts and where/when it becomes processed, so that unprocessed may be edited and already processed input may not be.

3. The input may be different or at least have different "looks" (echoing) than the output. Examples, raising lower case to upper case; confirmation by carriage return may be replaced by "Yes" in the output stream, etc. How do we handle this?

4. Other functionality of TTYIN which at the time of this writing I am not yet familar yet.

5. To what level do we support display stream graphic operations, such as MOVETO, DRAWLINE, clippingregion, XY coordinates, etc. Do we just paint graphics and don't capture them in any way, do we create imageobjects, or what?

6. Should we support two kinds of EXEC, one which uses the display stream to insure that all old programs will work as before, and a new type of EXEC which uses TEDITSTREAMS.

7. Do we implement this in one window in which it operates as one TEDITSTREAM, or do use a readonly TEDITSTREAM for backing, and use a separate window below the TEDIT window (with no border) to contain the current text which would use TTYIN to do the editing.

One advantage to implementing it as one TEDITSTREAM is that we could provide one standard method of backing display streams and handling text (ie, change the standard TEDIT interface to work this way) and possibly eliminate duplication of code such as TTYIN, thus making the system easier to maintain. On the other hand, it may turn out, that trying to handle this case as a general case in TEDIT is too envolved or too slow, and keeping them in two separate windows makes more sense.

Venue *MEDLEY LANGUAGE REFERENCE*

Address comments to:
Venue
User Documentation
1549 Industrial Road
San Carlos, CA 94070
415-508-9672

MEDLEY REFERENCE MANUAL

VOLUME I: LANGUAGE

April, 1993

Text was written and produced with Venue text formatting tools; PostScript printers were used to produce masters. The typeface is Palatino.

1. INTRODUCTION

Medley is a *programming system* that consists of a programming *language*, a large number of predefined programs (or *functions*) that you can use directly or as subroutines, and an *environment* that supports you with a variety of specialized programming tools. The language and predefined functions of Lisp are rich, but similar to those of other modern programming languages. The Medley programming environment, on the other hand, is very distinctive. Its main feature is an integrated set of programming tools that know enough about Interlisp and Common Lisp to act as semi-autonomous, intelligent "assistants" to you. This environment provides a completely self-contained world for creating, debugging and maintaining Lisp programs.

This manual describes all three parts of Medley. There are discussions of the language, about the pieces of the system that can be incorporated into your programs, and about the environment. The line between your code and the environment is thin and changing. Most users extend the environment with some special features of their own. Because Medley is so easily extended, the system has grown over time to incorporate many different ideas about effective and useful ways to program. This gradual accumulation over many years has resulted in a rich and diverse system. It is also the reason this manual is so large.

The rest of this manual describes the individual pieces of Medley; this chapter describes system as a whole—including the otherwise-unstated philosophies that tie it all together. It will give you a global view of Medley.

Lisp as a Programming Language

This manual is not an introduction to programming in Lisp. This section highlights a few key points about lisp that will make the rest of the manual clear.

In Lisp, large programs (or functions) are built up by composing the results of smaller ones. Although Medley, like most modern Lisps, lets you program in almost any style you can imagine, the natural style of Lisp is functional and recursive—each function computes its result by calling lower-level “building-block” functions, then passing that result back to its caller (rather than by producing “side-effects” on external data structures, for example).

Lisp is also a list-manipulation language. Like other languages, Lisp can process characters and numbers. But you get more power if you program at a higher level. The primitive data objects of Lisp are “atoms” (symbols or identifiers) and “lists” (sequences of atoms or lists), which you use to represent information and relationships. Each Lisp dialect has a set of operations that act on atoms and lists, and these operations comprise the core of the language.

Invisible in the programs, but essential to the Lisp style of programming, is an automatic memory management system (an “allocator” and a “garbage collector”). New storage is allocated automatically whenever a you create a new data object. And that storage is automatically reclaimed for reuse when no other object refers to it. Automated memory management is essential for rapid, large-scale program development because it frees you from the task of maintaining the details of memory administration, which change constantly during rapid program evolution.

A key property of Lisp is that Lisp function definitions are just pieces of Lisp list data. Each subfunction "call" (or *function application*) is written as a list with the function first, followed by its arguments. Thus, `(PLUS 1 2)` represents the expression `1+2`. A function's definition, then, is just a list of such function applications, to be evaluated in order. This representation of program as data lets you use the same operations on programs that you use on data—making it very easy to write Lisp programs that look at and change *other Lisp programs*. This, in turn, makes it easy to develop programming tools and translators, which was essential to the development of the Medley environment.

The most important benefit of this is that you can extend the Lisp programming language itself. Do you miss some favorite programming idiom? Just define a function that translates the desired expression into simpler Lisp. Now your idiom is *part of the language*. Medley has extensive facilities for making this type of language extension. Using this ability to extend itself, Interlisp has incorporated many of the constructs that have been developed in other modern programming languages (e.g. if-then-else, do loops, etc.).

Medley as an Interactive Environment

Medley programs should not be thought of as simple files of source code. All Medley programming takes place within the Medley environment, which is a completely self-sufficient environment for developing and using Medley programs. Beyond the obvious programming facilities (e.g., program editors, compilers, debuggers, etc.), the envionrment also contains a variety of tools that "keep track" of what happens. For example, the Medley File Manager notices when programs or data have been changed, so the system will know what needs to be saved at the end of a session. The "residential" style, where you stay inside the environment throughout the development, is essential for these tools to operate. Furthermore, this same environment is available to support the final production version, some parts providing run time support and other parts being ignored until the need arises for further debugging or development.

For terminal interaction, Medley provides a top level "Read-Eval-Print" executive, which reads whatever you type in, evaluates it, and prints the result. (This interaction is also recorded, so you can ask to do an action again, or even to undo the effects of a previous action.) Although Executives understand some specialized commands, most of the interaction will consist of simple Lisp expressions. So rather than special commands for operations like manipulating your files, you just type the same expressions that you would use to accomplish them in a Lisp program. This creates a very rich, simple, and uniform set of interactive commands, since any Lisp expression can be typed at an executive and evaluated immediately.

In normal use, you write a program (or rather, "define a function") by typing in an expression that invokes the "function defining" function `(DEFINEQ)`, giving it the name of the function being defined and its new definition. The newly-defined function can be executed immediately, simply by using it in a Lisp expression.

In addition to these basic programming tools, Medley also provides a wide variety of programming support mechanisms:

List structure editor	Since Lisp programs are represented as list structure, Medley provides an editor which allows one to change the list structure of a function's definition directly. See Chapter 16.
Pretty-printer	The pretty printer is a function that prints Lisp function definitions so that their syntactic structure is displayed by the indentation and fonts used. See page Chapter 26.
Debugger	When errors occur, the debugger is called, allowing you to examine and modify the context at the point of the error. Often, this lets you continue execution without starting from the beginning. Within a break, the full power of Interlisp is available to you. Thus, the broken function can be edited, data structures can be inspected and changed, other computations carried out, and so on. All of this occurs in the context of the suspended computation, which remains available to be resumed. See Chapter 14.
DWIM	The "Do What I Mean" package automatically fixes misspellings and errors in typing. See Chapter 20.
Programmer's Assistant	Medley keeps track of your actions during a session and allows each one to be replayed, undone, or altered. See Chapter 13.
Masterscope	Masterscope is a program analysis and management tool which can analyze users' functions and build (and automatically maintain) a data base of the results. This allows you to ask questions like "`WHO CALLS ARCTAN`" or "`WHO USES COEF1 FREELY`" or to request systematic changes like "`EDIT WHERE ANY` [function] `FETCHES ANY FIELD OF` [the data structure] `FOO`". See Chapter 19.
Record / Datatype Package	Medley allows you to define new data structures. This enables one to separate the issues of data access from the details of how the data is actually stored. See Chapter 8.
File Manager	Source code files in Medley are managed by the system, removing the problem of ensuring timely file updates from the user. The file manager can be modified and extended to accomodate new types of data. See Chapter 17.
Performance Analysis	These tools allow statistics on program operation to be collected and analyzed. See Chapter 22.
Multiple Processes	Multiple and independent processes simplify problems which require logically separate pieces of code to operate in parallel. See Chapter 23.

Windows The ability to have multiple, independent windows on the display allows many different processes or activities to be active on the screen at once. See Chapter 28.

Inspector The inspector is a display tool for examining complex data structures encountered during debugging. See Chapter 26.

These facilities are tightly integrated, so they know about and use each other, just as they can be used by user programs. For example, Masterscope uses the structural editor to make systematic changes. By combining the program analysis features of Masterscope with the features of the structural editor, large scale system changes can be made with a single command. For example, when the lowest-level interface of the Medley I/O system was changed to a new format, the entire edit was made by a single call to Masterscope of the form `EDIT WHERE ANY CALLS '(BIN BOUT ...)`. [Burton et al., 1980] This caused Masterscope to invoke the editor at each point in the system where any of the functions in the list `'(BIN BOUT ...)` were called. This ensured that no functions used in input or output were overlooked during the modification.

Philosophy

Medley's extensive environmental support has developed over the years to support a particular style of programming called "exploratory programming" [Sheil, 1983]. For many complex programming problems, the task of program creation is *not* simply one of writing a program to fulfill specifications. Instead, it is a matter of exploring the problem (trying out various solutions expressed as partial programs) until one finds a good solution (or sometimes, any solution at all!). Such programs are by nature evolutionary; they are transformed over time from one realization to another in response to a growing understanding of the problem. This point of view has lead to an emphasis on having the tools available to analyze, alter, and test programs easily. One important aspect of this is that the tools be designed to work together in an integrated fashion, so that knowledge about the user's programs, once gained, is available throughout the environment.

The development of programming tools to support exploratory programming is itself an exploration. No one knows all the tools that will eventually be found useful, and not all programmers want all of the tools to behave the same way. In response to this diversity, Interlisp has been shaped, by its implementors and by its users, to be easily extensible in several different ways. First, there are many places in the system where its behavior can be adjusted by the user. One way that this can be done is by changing the value of various "flags" or variables whose values are examined by system code to enable or suppress certain behavior. The other is where the user can provide functions or other behavioral specifications of what is to happen in certain contexts. For example, the format used for each type of list structure when it is printed by the pretty-printer is determined by specifications that are found on the list `PRETTYPRINTMACROS`. Thus, this format can be changed for a given type simply by putting a printing specification for it on that list.

Another way in which users can affect Medley's behavior is by redefining or changing system functions. The "Advise" capability, for instance, lets you modify the operation of virtually any function in the system by wrapping code "around" the selected function. (This same philosophy extends to breaking and tracing, so almost any function in the system can be broken or traced.) Since the entire system is implemented in Lisp, there are few places where the system's behavior depends on anything that you can't modify (such as a low level system implementation language).

While these techniques provide a fair amount of tailorability, there's a price: Medley is complex. There are many flags, parameters, and controls that affect its behavior. Because of this complexity, Interlisp tends to be more comfortable for experts, rather than casual users. Beginning users of Interlisp should depend on the default settings of parameters until they learn what dimensions of flexibility are available. At that point, they can begin to "tune" the system to their preferences.

Appropriately enough, even Medley's underlying philosophy was itself discovered during Medley's development, rather than laid out beforehand. The Medley environment and its interactive style were first analyzed in Sandewall's excellent paper [Sandewall, 1978]. The notion of "exploratory programming" and the genesis of the Interlisp programming tools in terms of the characteristic demands of this style of programming was developed in [Sheil, 1983]. The evolution and structure of the Interlisp programming environment are discussed in greater depth in [Teitelman & Masinter, 1981].

How to Use this Manual

This document is a reference manual, not a primer. We have tried to provide a manual that is complete, and that lets you find particular items as easily as possible. Sometimes, these goals have been achieved at the expense of simplicity. For example, many functions have a number of arguments that are rarely used. In the interest of providing a complete reference, these arguments are fully explained, even though you will normally let them default. There is a lot of information in this manual that is of interest only to experts.

Do not try to read straight through this manual, like a novel. In general, the chapters are organized with overview explanations and the most useful functions at the beginning of the chapter, and implementation details towards the end. If you are interested in becoming acquainted with Medley, we urge you to work through *An Introduction to Medley* before attempting this manual.

A few comments about the notational conventions used in this manual:

Lisp object notation: All Interlisp objects in this manual are printed in the same font: Functions `(AND, PLUS, DEFINEQ, LOAD)`; Variables `(MAX.INTEGER, FILELST, DFNFLG)`; and arbitrary Interlisp expressions: `(PLUS 2 3)`, `(PROG ((A 1)) ...)`, etc.

Case is significant: *In Interlisp, upper and lower case is significant.* The variable `FOO` is not the same as the variable `foo` or the variable `Foo`. By convention, most Interlisp system functions and variables are all uppercase, but users are free to use upper and lower case for their own functions and variables as they wish.

One exception to the case-significance rule is provided by the CLISP facility, which lets you type iterative statements and record operations in either all uppercase or all lowercase letters: `(for X` from 1 to 5 ...) is the same as (FOR X FROM 1 TO 5 ...). The few situations where this is the case are explicitly mentioned in the manual. Generally, assume that case is significant.

This manual contains a large number of descriptions of functions, variables, commands, etc, which are printed in the following standard format:

(**FOO** *BAR BAZ*) [Function]

This is a description for the function named FOO. FOO has two arguments, *BAR* and *BAZ*. Some system functions have extra optional arguments that are not documented and should not be used. These extra arguments are indicated by "—".

The descriptor [Function] indicates that this is a function, rather than a [Variable], [Macro], etc. For function definitions only, this can also indicate whether the function takes a fixed or variable number of arguments, and whether the arguments are evaluated or not. [Function] indicates a lambda spread function (fixed number of arguments, evaluated), the most common type.

References

[Burton, et al., 1980] Burton, R. R., L. M. Masinter, A. Bell, D. G. Bobrow, W. S. Haugeland, R.M. Kaplan and B.A. Sheil, "Interlisp-D: Overview and Status" — in [Sheil & Masinter, 1983].

[Sandewall, 1978] Sandewall, Erik, "Programming in the Interactive Environmnet: The LISP Experience" — *ACM Computing Surveys*, vol 10, no 1, pp 35-72, (March 1978).

[Sheil, 1983] Sheil, B.A., "Environments for Exploratory Programming" — *Datamation*, (February, 1983) — also in [Sheil & Masinter, 1983].

[Sheil & Masinter, 1983] Sheil, B.A. and L. M. Masinter, "Papers on Interlisp-D", Xerox PARC Technical Report CIS-5 (Revised), (January, 1983).

[Teitelman & Masinter, 1981] Teitelman, W. and L. M. Masinter, "The Interlisp Programming Environment" — *Computer*, vol 14, no 4, pp 25-34, (April 1981) — also in [Sheil & Masinter, 1983].

A litatom (for "literal atom") is an object that conceptually consists of a print name, a value, a function definition, and a property list. Litatoms are also known as "symbols" in Common Lisp. For clarity, we will use the term "symbol".

A symbol is read as any string of non-delimiting characters that cannot be interpreted as a number. The syntactic characters that delimit symbols are called "separator" or "break" characters (see Chapter 25) and normally are space, end-of-line, line-feed, left parenthesis `(`, right parenthesis `)`, double quote `"`, left square bracket `[`, and right square bracket `]`. However, any character may be included in a symbol by preceding it with the character `%`. Here are some examples of symbols:

```
A wxyz 23SKIDDOO %]
Long% Litatom% With% Embedded% Spaces
```

(**LITATOM** *X*) [Function]

Returns `T` if *X* is a symbol, `NIL` otherwise. Note that a number is not a symbol.

```
(LITATOM NIL) = T
```

(**ATOM** *X*) [Function]

Returns `T` if *X* is an atom (i.e., a symbol or a number) or `NIL` (e.g. `(ATOM NIL) = T`); otherwise returns `NIL`.

Warning: `(ATOM X)` is `NIL` if `X` is an array, string, etc. In Common Lisp, the function `CL:ATOM` is defined equivalent to the Interlisp function `NLISTP`.

Each symbol has a print name, a string of characters that uniquely identifies that symbol: Those characters that are output when the symbol is printed using `PRIN1`, e.g., the print name of the symbol `ABC%(D` consists of the five characters `ABC(D`.

Symbols are unique: If two symbols print the same, they will always be `EQ`. Note that this is not true for strings, large integers, floating-point numbers, etc.; they all can print the same without being `EQ`. Thus, if `PACK` or `MKATOM` is given a list of characters corresponding to a symbol that already exists, they return a pointer to that symbol, and do not make a new symbol. Similarly, if the read program is given as input a sequence of characters for which a symbol already exists, it returns a pointer to that symbol.

Symbol names are limited to 255 characters. Attempting to create a larger symbol will cause an error: `Atom too long`.

Sometimes we'll refer to a "`PRIN2`-name". The `PRIN2`-name of a symbol is those characters output when it is printed using `PRIN2`. So the `PRIN2`-name of the symbol `ABC%(D` is the six characters `ABC%(D`. The `PRIN2`-name depends on what readtable is being used (see Chapter 25), since this determines where `%`s will be inserted. Many of the functions below allow either print names or `PRIN2`-names to be used, as specified by `FLG` and `RDTBL` arguments. If `FLG` is `NIL`, print names are used. Otherwise, `PRIN2`-names are used, computed with respect to the readtable `RDTBL` (or the current readtable, if `RDTBL` = `NIL`).

(**MKATOM** *X*) [Function]

Creates and returns a symbol whose print name is the name as that of the string *X* or, if *X* is not a string, the same as that of (MKSTRING X). Examples:

```
(MKATOM '(A B C)) => %(A% B% C%)
(MKATOM "1.5") => 1.5
```

Note that the last example returns a number, not a symbol. It is a deeply-ingrained feature of Interlisp that no symbol can have the print name of a number.

(**SUBATOM** *X N M*) [Function]

Returns a symbol made from the *N*th through *M*th characters of the print name of *X*. If *N* or *M* are negative, they specify positions counting backwards from the end of the print name. Equivalent to (MKATOM (SUBSTRING X N M)). Examples:

```
(SUBATOM "FOO1.5BAR" 4 6) => 1.5
(SUBATOM '(A B C) 2 -2) => A% B% C
```

(**PACK** *X*) [Function]

If *X* is a list of symbols, PACK returns a single symbol whose print name is the concatenation of the print names of the symbols in *X*. If the concatenated print name is the same as that of a number, PACK returns that number. For example:

```
(PACK '(A BC DEF G)) => ABCDEFG
(PACK '(1 3.4)) => 13.4
(PACK '(1 E -2)) => .01
```

Although X is usually a list of symbols, it can be a list of arbitrary objects. The value of PACK is still a single symbol whose print name is the concatenation of the print names of all the elements of *X*, e.g.,

```
(PACK '((A B) "CD")) => %(A% B%)CD
```

If *X* is not a list or NIL, PACK generates the error Illegal arg.

(**PACK*** X_1 X_2... X_N) [NoSpread Function]

Version of PACK that takes an arbitrary number of arguments, instead of a list. Examples:

```
(PACK* 'A 'BC 'DEF 'G => ABCDEFG
(PACK* 1 3.4)) => 13.4
```

(**GENSYM** *PREFIX* — — — —) [Function]

Returns a symbol of the form Xnnnn, where X = *PREFIX* (or A if *PREFIX* is NIL) and nnnn is an integer. Thus, the first one generated is A0001, the second A0002, etc. The integer suffix is always at least four characters long, but it can grow beyond that. For example, the next symbol produced after A9999 would be A10000. GENSYM provides a way of generating symbols for various uses within the system .

Note: The Common Lisp function CL:GENSYM is not the same as Interlisp's GENSYM. Interlisp always creates interned symbols whereas CL:GENSYM creates uninterned symbols.

GENNUM [Variable]

The value of GENNUM, initially 0, determines the next GENSYM, e.g., if GENNUM is set to 23, (GENSYM) = A0024.

The term "gensym" is used to indicate a symbol that was produced by the function GENSYM. Symbols generated by GENSYM are the same as any other symbols: they have property lists, and can be given function definitions. The symbols are not guaranteed to be new. For example, if the user has previously created A0012, either by typing it in, or via PACK or GENSYM itself, then if GENNUM is set to 11, the next symbol returned by GENSYM will be the A0012 already in existence.

(**MAPATOMS** *FN*) [Function]

Applies *FN* (a function or lambda expression) to every symbol in the system. Returns NIL. For example:

```
(MAPATOMS (FUNCTION (LAMBDA(X) (if (GETD X) then (PRINTX)]
```

will print every symbol with a function definition.

Warning: Be careful if *FN* is a lambda expression or an interpreted function: since NOBIND is a symbol, it will eventually be passed as an argument. The first reference to that argument within the function will signal an error.

A way around this problem is to use a Common Lisp function, so that the Common Lisp interpreter will be invoked. It will treat the argument as local, not special and no error will be signaled. An alternative solution is to include the argument to the Interlisp function in a LOCALVARS declaration and then compile the function before passing it to MAPATOMS. This will significantly speed up MAPATOMS.

(**APROPOS** *STRING ALLFLG QUITFLG OUTPUT*) [Function]

APROPOS scans all symbols in the system for those which have *STRING* as a substring and prints them on the terminal along with a line for each relevant item defined for each selected symbol. Relevant items are:

- function definitions, for which only the arglist is printed
- dynamic variable values
- non-null property lists

PRINTLEVEL (see Chapter 25) is set to (3 . 5) when APROPOS is printing.

If *ALLFLG* is NIL, then symbols with no relevant items and "internal" symbols are omitted ("internal" currently means those symbols whose print name begins with a \ or those symbols produced by GENSYM). If *ALLFLG* is a function, it is used as a predicate on symbols selected by the substring match, with value NIL meaning to omit the symbol. If *ALLFLG* is any other non-NIL value, then no symbols are omitted.

Note: Unlike CL:APROPOS which lets you designate the package to search, APROPOS searches *all* packages.

Using Symbols as Variables

Symbols are commonly used as variable names. Each symbol has a "top level" value, which can be an arbitrary object. Symbols may also be given special variable bindings within PROGs or functions, which only exist for the duration of the function. When a symbol is evaluated, the "current" variable binding is returned. This is the most recent special variable binding, or the top-level binding if the symbol hasn't been rebound. SETQ is used to change the current binding. For more information on variable bindings in Interlisp, see Chapter 11.

A symbol whose top-level value is the symbol NOBIND is considered to have no value. If a symbol has no local bindings, and its top-level value is NOBIND, trying to evaluate it will cause an unbound-atom error. In addition, if a symbol's local binding is to NOBIND, trying to evaluate it will cause an error.

The symbols T and NIL always evaluate to themselves. Attempting to change the value of T or NIL with the functions below will generate the error; `Attempt to set T` or `Attempt to set NIL`.

The following functions (except BOUNDP) will also generate the error `Arg not litatom`, if not given a symbol.

(**BOUNDP** *VAR*) [Function]

Returns T if *VAR* has a special variable binding, or if *VAR* has a top-level value other than NOBIND; otherwise NIL. That is, if X is a symbol, `(EVAL X)` will cause an `Unbound atom` error if and only if `(BOUNDP X)` returns NIL.

Note: The Interlisp interpreter has been modified so that it will generate an `Unbound Variable` error when it encounters any symbol bound to NOBIND. This is a change from previous releases that only signaled an error when a symbol had a top-level binding of NOBIND in addition to no dynamic binding.

(**SET** *VAR VALUE*) [NoSpread Function]

Sets the "current" value of *VAR* to *VALUE*, and returns *VALUE*.

SET is a normal function, so both *VAR* and *VALUE* are evaluated before it is called. Thus, if the value of X is B, and value of Y is C, then `(SET X Y)` would result in B being set to C, and C being returned as the value of SET.

(**SETQ** *VAR VALUE*) [NoSpread Function]

Like SET, but *VAR* is not evaluated, *VALUE* is. Thus, if the value of X is B and the value of Y is C, `(SETQ X Y)` would result in X (not B) being set to C, and C being returned.

Actually, neither argument is evaluated during the calling process. However, SETQ itself calls EVAL on its second argument. As a result, typing `(SETQ VAR FORM)` and `SETQ (VAR FORM)` to the Interlisp Executive are equivalent: in both cases VAR is not evaluated, and FORM is.

(**SETQQ** *VAR VALUE*) [NoSpread Function]

Like SETQ, but neither argument is evaluated, e.g., `(SETQQ X (A B C))` sets X to `(A B C)`.

(**PSETQ** *VAR_1 $VALUE_1$ ... VAR_N $VALUE_N$*) [Macro]

Does a SETQ in parallel of VAR_1 (unevaluated) to $VALUE_1$, VAR_2 to $VALUE_2$, etc. All of the $VALUE_i$ terms are evaluated before any of the assignments. Therefore, (PSETQ A B B A) can be used to swap the values of the variables A and B.

(**GETTOPVAL** *VAR*) [Function]

Returns the top level value of *VAR* (even if NOBIND), regardless of any intervening local bindings.

(**SETTOPVAL** *VAR VALUE*) [Function]

Sets the top level value of *VAR* to *VALUE*, regardless of any intervening bindings, and returns *VALUE*.

(**GETATOMVAL** *VAR*) [Function]

Same as (GETTOPVAL VAR).

(**SETATOMVAL** *VAR VALUE*) [Function]

Same as SETTOPVAL.

Note: The compiler (see Chapter 18) treats variables somewhat differently from the interpreter, and you need to be aware of these differences when writing functions that will be compiled. For example, variable references in compiled code are not checked for NOBIND, so compiled code will not generate unbound-atom errors. In general, it is better to debug interpreted code, before compiling it for speed. The compiler offers some facilities to increase the efficiency of variable use in compiled functions: Global variables can be defined so that the entire stack is not searched at each variable reference. Local variables have bindings that are not visible outside the function, which reduces variable conflicts and makes variable lookup faster.

Function Definition Cells

Each symbol has a function-definition cell, which is accessed when that symbol is used as a function. This is described in detail in Chapter 10.

Property Lists

Each symbol has an associated property list, which allows a set of named objects to be associated with the symbol. A property list associates a name (known as a “property name” or “property”) with an arbitrary object (the “property value” or “value”). Sometimes the phrase “to store on the property X” is used, meaning to place the indicated information on a property list under the property name X.

Property names are usually symbols or numbers, although no checks are made. However, the standard property list functions all use EQ to search for property names, so they may not work with non-atomic property names. The same object can be used as both a property name and a property value.

Many symbols in the system already have property lists, with properties used by the compiler, the break package, DWIM, etc. Be careful not to clobber such system properties. The variable SYSPROPS is a list of property names used by the system.

The functions below are used to manipulate the property lists of symbols. Except when indicated, they generate the error *ATM* `is not a SYMBOL`, if given an object that is not a symbol.

(**GETPROP** *ATM PROP*) [Function]

Returns the property value for *PROP* from the property list of *ATM*. Returns `NIL` if *ATM* is not a symbol, or *PROP* is not found. `GETPROP` also returns `NIL` if there is an occurrence of *PROP* but the corresponding property value is `NIL`. This can be a source of program errors.

Note: `GETPROP` used to be called `GETP`.

(**PUTPROP** *ATM PROP VAL*) [Function]

Puts the property *PROP* with value *VAL* on the property list of *ATM*. *VAL* replaces any previous value for the property *PROP* on this property list. Returns *VAL*.

(**ADDPROP** *ATM PROP NEW FLG*) [Function]

Adds the value *NEW* to the list which is the value of property *PROP* on the property list of the *ATM*. If *FLG* is `T`, *NEW* is `CONS`ed onto the front of the property value of *PROP*; otherweise, it is `NCONC`ed on the end (using `NCONC1`). If *ATM* does not have a property *PROP*, or the value is not a list, then the effect is the same as `(PUTPROP ATM PROP (LIST NEW))`. `ADDPROP` returns the (new) property value. Example:

```
←(PUTPROP 'POCKET 'CONTENTS NIL)
 (NIL)
←(ADDPROP 'POCKET 'CONTENTS 'COMB)
 (COMB)
←(ADDPROP 'POCKET 'CONTENTS 'WALLET)
 (COMB WALLET)
```

(**REMPROP** *ATM PROP*) [Function]

Removes all occurrences of the property *PROP* (and its value) from the property list of *ATM*. Returns *PROP* if any were found (`T` if *PROP* is `NIL`), otherwise `NIL`.

(**CHANGEPROP** *X PROP1 PROP2*) [Function]

Changes the property name of property *PROP1* to *PROP2* on the property list of *X* (but does not affect the value of the property). Returns *X*, unless *PROP1* is not found, in which case it returns `NIL`.

(**PROPNAMES** *ATM*) [Function]

Returns a list of the property names on the property list of *ATM*.

(**DEFLIST** *L PROP*) [Function]

Used to put values under the same property name on the property lists of several symbols. *L* is a list of two-element lists. The first element of each is a symbol, and the second element is the property vqalue of the property *PROP*. Returns `NIL`. For example:

```
(DEFLIST '((FOO MA)(BAR CA)(BAZ RI)) 'STATE)
```

puts MA on FOO's STATE property, CA on BAR's STATE property, and RI on BAZ's STATE property.

Property lists are conventionally implemented as lists of the form

($NAME_1$ $VALUE_1$ $NAME_2$ $VALUE_2$...)

although the user can store anything as the property list of a symbol. However, thge functions which manipulate property lists observe this convention by searching down the property lists two CDRs at a time. Most of these functions also generate the error Arg not litatom if given an argument which is not a symbol, so they cannot be used directly on lists. (LISTPUT, LISTPUT1, LISTGET, and LISTGET1 are functions similar to PUTPROP and GETPROP that work directly on lists (see Chapter 3) . The property lists of symbols can be directly accessed with the following functions.

(**GETPROPLIST** *ATM*) [Function]

Returns the property list of *ATM*.

(**SETPROPLIST** *ATM LST*) [Function]

If *ATM* is a symbol, sets the property list of *ATM* to be *LST*, and returns *LST* as its value.

(**GETLIS** *X PROPS*) [Function]

Searches the property list of *X*, and returns the property list as of the first property on *PROPS* that it finds. For example:

```
←(GETPROPLIST 'X)
 (PROP1 A PROP3 B A C)
←(GETLIS 'X '(PROP2 PROP3))
 (PROP3 B A C)
```

Returns NIL if no element on props is found. *X* can also be a list itself, in which case it is searched as described above. If *X* is not a symbol or a list, returns NIL.

(**REMPROPLIST** *ATM PROPS*) [Function]

Removes all occurrences of all properties on the list *PROPS* (and their corresponding property values) from the property list of *ATM*. Returns NIL.

Print Names

The term "print name" has an extended meaning: The characters that are output when *any object* is printed. In Medley, all objects have print names, although only symbols and strings have their print names explicitly stored. Symbol print names are limited to 255 characters.

This section describes a set of functions that can be used to access and manipulate the print names of any object, though they are primarily used with the print names of symbols. In Medley, print functions qualify symbol names with a package prefix if the symbol is not accessible in the current package. The exception is Interlisp's PRIN1, which does not include a package prefix.

The print name of an object is those characters that are output when the object is printed using PRIN1, e.g., the print name of the list (A B "C") consists of the seven characters (A B C) (two of the characters are spaces).

The PRIN2-name of an object is those characters output when the object is printed using PRIN2. Thus the PRIN2-name of the list (A B "C") is the 9 characters (A B "C") (including the two spaces). The PRIN2-name depends on what readtable is being used (see Chapter 25), since this determines where %s will be inserted. Many of the functions below allow either print names of PRIN2-names to be used, as specified by FLG and RDTBL arguments. If FLG is NIL, print names are used. Otherwise, PRIN2-names are used, computed with respect to the readtable RDTBL (or the current readtable, if RDTBL = NIL).

The print name of an integer depends on the setting of RADIX (see Chapter 25). The functions described in this section (UNPACK, NCHARS, etc.) define the print name of an integer as though the radix was 10, so that (PACK (UNPACK 'X9)) will always be X9 (and not X11, if RADIX is set to 8). However, integers will still be printed by PRIN1 using the current radix. The user can force these functions to use print names in the current radix by changing the setting of the variable PRXFLG (see Chapter 25).

(**CL:SYMBOL-NAME** *SYM*) [Common Lisp Function]

Returns a string displaced to the *SYM* print name. Strings returned from CL:SYMBOL-NAME may be destructively modified without affecting *SYM*'s print name.

(**NCHARS** *X FLG RDTBL*) [Function]

Returns the number of characters in the print name of *X*. If *FLG* = T, the PRIN2-name is used. Examples:

```
(NCHARS 'ABC) => 3
(NCHARS "ABC" T) => 5
```

NCHARS works most efficiently on symbols and strings, but can be given any object.

(**NTHCHAR** *X N FLG RDTBL*) [Function]

Returns *X*, if *X* is a tail of the list Y; otherwise NIL. *X* is a tail of Y if it is EQ to 0 or more CDRs of Y.

```
(NTHCHAR 'ABC 2)  =>  B
(NTHCHAR 15.6 2)  =>  5
(NTHCHAR 'ABC%(D -3 T)  =>  %%
(NTHCHAR "ABC" 2)  =>  B
(NTHCHAR "ABC" 2 T)  =>  A
```

NTHCAR and NCHARS work much faster on objects that actually have an internal representation of their print name, i.e., symbols and strings, than they do on numbers and lists, since they don't have to simulate printing.

(**L-CASE** *X FLG*) [Function]

Returns a lowercase version of *X*. If *FLG* is T, the first letter is capitalized. If *X* is a string, the value of L-CASE is also a string. If *X* is a list, L-CASE returns a new list in which L-CASE is computed for each corresponding element and non-NIL tail of the original list. Examples:

```
(L-CASE 'FOO) => foo
(L-CASE 'FOO T) => Foo
(L-CASE "FILE NOT FOUND" T) => "File not found"
```

```
(L-CASE '(JANUARY FEBRUARY (MARCH "APRIL")) T) =>
   '(January February (March "April"))
```

(**U-CASE** *X*) [Function]

Like L-CASE, but returns the uppercase version of *X*.

(**U-CASEP** *X*) [Function]

Returns T if *X* contains no lowercase letters; NIL otherwise.

Characters and Character Codes

Characters are represented 3 different ways in Medley. In Interlisp they are single-character symbols or integer character codes. In Common Lisp they are instances of the CHARACTER datatype. In general Interlisp character functions don't accept Common Lisp characters and vice versa. The only exceptions are Interlisp string-manipulation functions that accept "string or symbol" types as arguments.

You can convert between Interlisp and Common Lisp characaters by using the functions CL:CODE-CHAR, CL:CHAR-CODE, and CHARCODE (see below).

Medley uses the 16-bit NS character set, described in the document Character Code Standard (Xerox System Integration Standards, XSIS 058404, April 1984). Legal character codes range from 0 to 65535. The NS (Network Systems) character encoding encompasses a much wider set of available characters than the 8-bit character standards (such as ASCII), including characters comprising many foreign alphabets and special symbols. For instance, Medley supports the display and printing of the following:

- Le système d'information Medley est remarqueablement polyglotte
- Das Medley Kommunikationssystem bietet merkwürdige multilinguale Nutzmöglichkeiten
- M ⊆ ❑ [w] ⇔ ∀ v with Rwv: M ⊆ [v]

These characters can be used in strings, symbol print names, symbolic files, or anywhere else 8-bit characters could be used. All of the standard string and print name functions (RPLSTRING, GNC, NCHARS, STRPOS, etc.) accept symbols and strings containing NS characters. For example:

```
←(STRPOS "char""this is an 8-bit character string")
   18
←(STRPOS "char""celui-ci comports des characteres NS")
   23
```

In almost all cases, a program does not have to distinguish between NS characters or 8-bit characters. The exception to this rule is the handling of input/output operations (see Chapter 25).

The function CHARCODE (see below) provides a simple way to create individual NS character codes. The VirtualKeyboards library module provides a set of virtual keyboards that allows keyboard or mouse entry of NS characters.

(**PACKC** *X*) [Function]

Like PACK except *X* is a list of character codes. For example,

```
(PACKC '(70 79 79))  => FOO
```

(**CHCON** *X FLG RDTBL*) [Function]

Like UNPACK, but returns the print name of *X* as a list of character codes. If *FLG* = T, the PRIN2-name is used. For example:

```
(CHCON 'FOO)  => (70 79 79)
```

(**DCHCON** *X SCRATCHLIST FLG RDTBL*) [Function]

Like DUNPACK.

(**NTHCHARCODE** *X N FLG RDTBL*) [Function]

Like NTHCHAR, but returns the character code of the *N*th character of the print name of *X*. If *N* is negative, it is interpreted as a count backwards from the end of *X*. If the absolute value of *N* is greater than the number of characters in *X*, or 0, then the value of NTHCHARCODE is NIL.

If *FLG* is T, then the PRIN2-name of *X* is used, computed with respect to the readtable.

(**CHCON1** *X*) [Function]

Returns the character code of the first character of the print name of *X*; equal to (NTHCHARCODE X 1).

(**CHARACTER** *N*) [Function]

N is a character code. Returns the symbol having the corresponding single character as its print name.

```
(CHARACTER 70) => F
```

(**FCHARACTER** *N*) [Function]

Fast version of CHARACTER that compiles open.

The following function makes it possible to gain the efficiency that comes from dealinig with character codes without losing the symbolic advantages of character symbols.

(**CHARCODE** *CHAR*) [Function]

Returns the character code specified by *CHAR* (unevaluated). If *CHAR* is a one-character symbol or string, the corresponding character code is simply returned. Thus, (CHARCODE A) is 65, (CHARCODE 0) is 48. If *CHAR* is a multi-character symbol or string, it specifies a character code as described below. If *CHAR* is NIL, CHARCODE simply returns NIL. Finally, if *CHAR* is a list structure, the value is a copy of *CHAR* with all the leaves replaced by the corresponding character codes. For instance, (CHARCODE (A (B C))) => (65 (66 67)).

If a character is specified by a multi-character symbol or string, CHARCODE interprets it as follows:

CR, SPACE, etc.

The variable CHARACTERNAMES contains an association list mapping special symbols to character codes. Among the characters defined this way are CR (13), LF (10), SPACE or SP (32), ESCAPE or ESC (27), BELL (7), BS (8), TAB (9), NULL (0), and DEL (127). The symbol EOL maps into the appropriate end-of-line character code in the different Interlisp implementations (31 in Interlisp-10, 13 in Interlisp-D, 10 in Interlisp-VAX). Examples:

```
(CHARCODE SPACE)  =>  32
(CHARCODE CR)  =>  13
```

CHARSET, CHARNUM, CHARSET-CHARNUM

If the character specification is a symbol or string of the form CHARSET, CHARNUM, or CHARSET-CHARNUM, the character code for the character number CHARNUM in the character set CHARSET is returned.

The 16-bit NS character encoding is divided into a large number of “character sets”. Each 16-bit character can be decoded into a character set (an integer from 0 to 254 inclusive) and a character number (also an integer from 0 to 254 inclusive). CHARSET is either an octal number, or a symbol in the association list CHARACTERSETNAMES (which defines the character sets for GREEK, CYRILLIC, etc.).

CHARNUM is either an octal number, a single-character symbol, or a symbol from the association list CHARACTERNAMES. If CHARNUM is a single-digit number, it is interpreted as the character “2”, rather than as the octal number 2. Examples:

```
(CHARCODE 12,6)  =>  2566
(CHARCODE 12,SPACE)  =>  2592
(CHARCODE GREEK,A) => 9793
```

↑CHARSPEC (control chars)

If the character specification is a symbol or string of one of the forms above, preceded by the character ↑, this indicates a “control character,” derived from the normal character code by clearing the seventh bit of the character code (normally set). Examples:

```
(CHARCODE ↑A)  =>  1
(CHARCODE ↑GREEK,A)  =>  9729
```

#CHARSPEC (meta chars)

If the character specification is a symbol or string of one of the forms above, preceded by the charactger #, this indicates a meta character, derived from the normal character code by setting the eighth bit of the character code (normally cleared). ↑ and # can both be set at once. Examples:

```
(CHARCODE #A)  =>  193
(CHARCODE #↑GREEK,A)  =>  9857
```

A CHARCODE form can be used wherever a structure of character codes would be appropriate. For example:

```
(FMEMB (NTHCHARCODE X 1)(CHARCODE (CR LF SPACE ↑A)))
(EQ (READCCODE FOO)(CHARCODE GREEK,A))
```

There is a macro for CHARCODE which causes the character-code structure to be constructed at compile-time. Thus, the compiled code for these examples is exactly as efficient as the less readable:

```
(FMEMB (NTHCHARCODE X 1)(QUOTE (13 10 32 1)))
(EQ (READCCODE FOO)9793)
```

(**CL:CHAR-CODE** *CHAR*) [Common Lisp Function]

Returns the Interlisp character code of *CHAR*. Use to convert a Common Lisp character to an Interlisp character code.

(**CL:CODE-CHAR** *N*) [Common Lisp Function]

Returns a character with the given non-negative integer *N* code. Returns NIL if no character is possible with *N*. Use to convert an Interlisp character code to a Common Lisp character.

(**SELCHARQ** *E* $CLAUSE_1$... $CLAUSE_N$ *DEFAULT*) [Function]

Lets you branch one of several ways, based on the character code *E*. The first item in each $CLAUSE_N$ is a character code or list of character codes, given in the form CHARCODE would accept. If the value of *E* is a character code or NIL, and it is EQ or MEMB to the result of applying CHARCODE to the first element of a clause, the remaining forms of that clause are evaluated. Otherwise, the default is evaluated.

Thus

```
(SELCHARQ (BIN FOO))
     ((SPACE TAB)(FUM))
     ((↑D NIL)(BAR))
     (a (BAZ))
     (ZIP)))
```

is exactly equivalent to

```
(SELECTQ (BIN FOO))
     ((32 9)(FUM))
     ((4 NIL)(BAR))
     (97 (BAZ))
     (ZIP)))
```

If (BIN FOO) returned 32 (the SPACE character), the function FUM would be called.

[This page intentionally left blank]

TABLE of CONTENTS

Volume 1 - Lanuage Reference

Volume 2 - Environment Reference

Volume 3 - I/O Reference

[This page intentionally left blank]

4. STRINGS

A string represents a sequence of characters. Interlisp strings are a subtype of Common Lisp strings. Medley provides functions for creating strings, concatenating strings, and creating sub-strings of a string; all accepting or producing Common Lisp-acceptable strings.

A string is typed as a double quote (`"`), followed by a sequence of any characters except double quote and `%`, terminated by a double quote. To include `%` or `"` in a string, type `%` in front of them:

```
"A string"
"A string with %" in it, and a %%."
""       ; an empty string
```

Strings are printed by `PRINT` and `PRIN2` with initial and final double quotes, and `%`s inserted where necessary for it to read back in properly. Strings are printed by `PRIN1` without the double quotes and extra `%`s. The null string is printed by `PRINT` and `PRIN2` as `""`. `(PRIN1 "")` doesn't print anything.

Internally, a string is stored in two parts: a "string header" and the sequence of characters. Several string headers may refer to the the same character sequence, so a substring can be made by creating a new string header, without copying any characters. Functions that refer to "strings" actually manipulate string headers. Some functions take an "old string" argument, and re-use the string pointer.

(**STRINGP** *X*) [Function]

Returns *X* if *X* is a string, `NIL` otherwise.

(**STREQUAL** *X Y*) [Function]

Returns `T` if *X* and *Y* are both strings and they contain the same sequence of characters, otherwise `NIL`. `EQUAL` uses `STREQUAL`. Note that strings may be `STREQUAL` without being `EQ`. For instance,

```
(STREQUAL "ABC" "ABC")  =>  T
(EQ "ABC" "ABC")  =>  NIL
```

`STREQUAL` returns `T` if *X* and *Y* are the same string pointer, or two different string pointers which point to the same character sequence, or two string pointers which point to different character sequences which contain the same characters. Only in the first case would *X* and *Y* be `EQ`.

(**STRING-EQUAL** *X Y*) [Function]

Returns `T` if *X* and *Y* are either strings or symbols, and they contain the same sequence of characters, ignoring case. For instance,

```
(STRING-EQUAL "FOO" "Foo")  =>  T
(STRING-EQUAL "FOO" 'Foo)  =>  T
```

This is useful for comparing things that might want to be considered "equal" even though they're not both symbols in a consistent case, such as file names and user names.

(**STRING.EQUAL** *X Y*) [Function]

Returns T if the print names of *X* and *Y* contain the same sequence of characters, ignoring case. For instance,

```
(STRING-EQUAL "320" 320)  =>  T
(STRING-EQUAL "FOO" 'Foo)  =>  T
```

This is like STRING-EQUAL, but handles numbers, etc., where STRING-EQUAL doesn't.

(**ALLOCSTRING** *N INITCHAR OLD FATFLG*) [Function]

Creates a string of length *N* characters of *INITCHAR* (which can be either a character code or something coercible to a character). If *INITCHAR* is NIL, it defaults to character code 0. if *OLD* is supplied, it must be a string pointer, which is modified and returned.

If *FATFLG* is non-NIL, the string is allocated using full 16-bit NS characters (see Chapter 2) instead of 8-bit characters. This can speed up some string operations if NS characters are later inserted into the string. This has no other effect on the operation of the string functions.

(**MKSTRING** *X FLG RDTBL*) [Function]

If *X* is a string, returns *X*. Otherwise, creates and returns a string containing the print name of *X*. Examples:

```
(MKSTRING "ABC")  =>  "ABC"
(MKSTRING '(A B C))  =>  "(A B C)"
(MKSTRING NIL)  =>  "NIL"
```

Note that the last example returns the string "NIL", not the symbol NIL.

If *FLG* is T, then the PRIN2-name of *X* is used, computed with respect to the readtable *RDTBL*. For example,

```
(MKSTRING "ABC" T)  =>  "%"ABC%""
```

(**NCHARS** *X FLG RDTBL*) [Function]

Returns the number of characters in the print name of *X*. If *FLG*=T, the PRIN2-name is used. For example,

```
(NCHARS 'ABC)  =>  3
(NCHARS "ABC" T)  =>  5
```

Note: NCHARS works most efficiently on symbols and strings, but can be given any object.

(**SUBSTRING** *X N M OLDPTR*) [Function]

Returns the substring of *X* consisting of the *N*th through *M*th characters of *X*. If *M* is NIL, the substring contains the *N*th character thru the end of *X*. *N* and *M* can be negative numbers, which are interpreted as counts back from the end of the string, as with NTHCHAR (Chapter 2). SUBSTRING returns NIL if the substring is not well defined, (e.g., *N* or *M* specify character positions outside of *X*, or *N* corresponds to a character in *X* to the right of the character indicated by *M*). Examples:

```
(SUBSTRING "ABCDEFG" 4 6)  =>  "DEF"
(SUBSTRING "ABCDEFG" 3 3)  =>  "C"
(SUBSTRING "ABCDEFG" 3 NIL)  =>  "CDEFG"
(SUBSTRING "ABCDEFG" 4 -2)  =>  "DEF"
(SUBSTRING "ABCDEFG" 6 4)  =>  NIL
(SUBSTRING "ABCDEFG" 4 9)  =>  NIL
```

If *X* is not a string, it is converted to one. For example,

```
(SUBSTRING '(A B C) 4 6)  =>  "B C"
```

`SUBSTRING` does not actually copy any characters, but simply creates a new string pointer to the characters in *X*. If *OLDPTR* is a string pointer, it is modified and returned.

(**GNC** *X*) [Function]

"Get Next Character." Returns the next character of the string *X* (as a symbol); also removes the character from the string, by changing the string pointer. Returns `NIL` if *X* is the null string. If *X* isn't a string, a string is made. Used for sequential access to characters of a string. Example:

```
←(SETQ FOO "ABCDEFG")
  "ABCDEFG"
←(GNC FOO)
  A
←(GNC FOO)
  B
←FOO
  "CDEFG"
```

Note that if *A* is a substring of `B`, `(GNC A)` does not remove the character from `B`.

(**GLC** *X*) [Function]

"Get Last Character." Returns the last character of the string *X* (as a symbol); also removes the character from the string. Similar to `GNC`. Example:

```
←(SETQ FOO "ABCDEFG")
  "ABCDEFG"
←(GLC FOO)
  G
←(GLC FOO)
  F
←FOO
  "ABCDE"
```

(**CONCAT** X_1 X_2 ... X_N) [NoSpread Function]

Returns a new string which is the concatenation of (copies of) its arguments. Any arguments which are not strings are transformed to strings. Examples:

```
(CONCAT "ABC" 'DEF "GHI")  =>  "ABCDEFGHI"
(CONCAT '(A B C) "ABC")  =>  "(A B C)ABC"
```
`(CONCAT)` returns the null string, `""`

(**CONCATLIST** *L*) [Function]

L is a list of strings and/or other objects. The objects are transformed to strings if they aren't strings. Returns a new string which is the concatenation of the strings. Example:

```
(CONCATLIST '(A B (C D) "EF"))  =>  "AB(C D)EF"
```

(**RPLSTRING** *X N Y*) [Function]

Replaces the characters of string *X* beginning at character position *N* with string *Y*. *X* and *Y* are converted to strings if they aren't already. *N* may be positive or negative, as with SUBSTRING. Characters are smashed into (converted) *X*. Returns the string *X*. Examples:

```
(RPLSTRING "ABCDEF" -3 "END")  =>  "ABCEND"
(RPLSTRING "ABCDEFGHIJK" 4 '(A B C))  =>  "ABC(A B C)K"
```

Generates an error if there is not enough room in *X* for *Y*, i.e., the new string would be longer than the original. If *Y* was not a string, *X* will already have been modified since RPLSTRING does not know whether *Y* will "fit" without actually attempting the transfer.

Warning: In some implementations of Interlisp, if *X* is a substring of Z, Z will also be modified by the action of RPLSTRING or RPLCHARCODE. However, this is not guaranteed to be true in all cases, so programmers should not rely on RPLSTRING or RPLCHARCODE altering the characters of any string other than the one directly passed as argument to those functions.

(**RPLCHARCODE** *X N CHAR*) [Function]

Replaces the *N*th character of the string *X* with the character code *CHAR*. *N* may be positive or negative. Returns the new *X*. Similar to RPLSTRING. Example:

```
(RPLCHARCODE "ABCDE" 3 (CHARCODE F))  =>  "ABFDE"
```

(**STRPOS** *PAT STRING START SKIP ANCHOR TAIL CASEARRAY BACKWARDSFLG*) [Function]

STRPOS is a function for searching one string looking for another. *PAT* and *STRING* are both strings (or else they are converted automatically). STRPOS searches *STRING* beginning at character number *START*, (or 1 if *START* is NIL) and looks for a sequence of characters equal to *PAT*. If a match is found, the character position of the first matching character in *STRING* is returned, otherwise NIL. Examples:

```
(STRPOS "ABC" "XYZABCDEF")  =>  4
(STRPOS "ABC" "XYZABCDEF" 5)  =>  NIL
(STRPOS "ABC" "XYZABCDEFABC" 5)  =>  10
```

SKIP can be used to specify a character in *PAT* that matches any character in *STRING*. Examples:

```
(STRPOS "A&C&" "XYZABCDEF" NIL '&)  =>  4
(STRPOS "DEF&" "XYZABCDEF" NIL '&)  =>  NIL
```

If *ANCHOR* is T, STRPOS compares *PAT* with the characters beginning at position *START* (or 1 if *START* is NIL). If that comparison fails, STRPOS returns NIL without searching any further down *STRING*. Thus it can be used to compare one string with some *portion* of another string. Examples:

```
(STRPOS "ABC" "XYZABCDEF" NIL NIL T)  =>  NIL
```

```
(STRPOS "ABC" "XYZABCDEF" 4 NIL T)  =>  4
```

If *TAIL* is T, the value returned by STRPOS if successful is not the starting position of the sequence of characters corresponding to *PAT*, but the position of the first character after that, i.e., the starting position plus (NCHARS *PAT*). Examples:

```
(STRPOS "ABC" "XYZABCDEFABC" NIL NIL NIL T)  =>  7
(STRPOS "A" "A" NIL NIL NIL T)  =>  2
```

If *TAIL* = NIL, STRPOS returns NIL, or a character position within *STRING* which can be passed to SUBSTRING. In particular, (STRPOS "" "") => NIL. However, if *TAIL* = T, STRPOS may return a character position outside of *STRING*. For instance, note that the second example above returns 2, even though "A" has only one character.

If *CASEARRAY* is non-NIL, this should be a casearray like that given to FILEPOS (Chapter 25). The casearray is used to map the string characters before comparing them to the search string.

If *BACKWARDSFLG* is non-NIL, the search is done backwards from the end of the string.

(**STRPOSL** *A STRING START NEG BACKWARDSFLG*) [Function]

STRING is a string (or is converted automatically to a string), *A* is a list of characters or character codes. STRPOSL searches *STRING* beginning at character number *START* (or 1 if *START* = NIL) for one of the characters in *A*. If one is found, STRPOSL returns as its value the corresponding character position, otherwise NIL. Example:

```
(STRPOSL '(A B C) "XYZBCD")  =>  4
```

If *NEG* = T, STRPOSL searches for a character *not* on *A*. Example:

```
(STRPOSL '(A B C) "ABCDEF" NIL T)  =>  4
```

If any element of *A* is a number, it is assumed to be a character code. Otherwise, it is converted to a character code via CHCON1. Therefore, it is more efficient to call STRPOSL with *A* a list of character *codes*.

If *A* is a bit table, it is used to specify the characters (see MAKEBITTABLE below)

If *BACKWARDSFLG* is non-NIL, the search is done backwards from the end of the string.

STRPOSL uses a "bit table" data structure to search efficiently. If *A* is not a bit table, it is converted to a bit table using MAKEBITTABLE. If STRPOSL is to be called frequently with the same list of characters, a considerable savings can be achieved by converting the list to a bit table *once*, and then passing the bit table to STRPOSL as its first argument.

(**MAKEBITTABLE** *L NEG A*) [Function]

Returns a bit table suitable for use by STRPOSL. *L* is a list of characters or character codes, *NEG* is the same as described for STRPOSL. If *A* is a bit table, MAKEBITTABLE modifies and returns it. Otherwise, it will create a new bit table.

Note: If *NEG* = T, STRPOSL must call MAKEBITTABLE whether *A* is a list *or* a bit table. To obtain bit table efficiency with *NEG*=T, MAKEBITTABLE should be called with *NEG*=T, and the resulting "inverted" bit table should be given to STRPOSL with *NEG*=NIL.

[This page intentionally left blank]

5. ARRAYS

An Interlisp array is a one-dimensional vector of objects. Arrays are generally created by the function ARRAY. By contrast, Common Lisp arrays can be multi-dimensional.

Note: Interlisp arrays and Common Lisp arrays are *not* the same types. Interlisp functions only accept Interlisp arrays and vice versa. There are no functions to convert between the two types.

(**ARRAY** *SIZE TYPE INIT ORIG* —) [Function]

Creates and returns a new array that holds *SIZE* objects of type *TYPE*. If *TYPE* is NIL, the array can contain any arbitrary Lisp datum. In general, *TYPE* may be any of the various field specifications that are legal in DATATYPE declarations (see Chapter 8): POINTER, FIXP, FLOATP, (BITS *N*), etc. Medley will, if necessary, choose an "enclosing" type if the given one is not supported; for example, an array of (BITS 3) may be represented by an array of (BITS 8).

INIT is the initial value for each element of the new array. If not specified, the array elements will be initialized with 0 (for number arrays) or NIL (all other types).

Arrays can have either 0-origin or 1-origin indexing, as specified by the *ORIG* argument; if *ORIG* is not specified, the default is 1.

Arrays of type FLOATP are stored unboxed. This increases the space and time efficiency of FLOATP arrays. If you want to use boxed floating point numbers, use an array of type POINTER instead of FLOATP.

(**ARRAYP** *X*) [Function]

Returns X if *X* is an array, NIL otherwise.

(**ELT** *ARRAY N*) [Function]

Returns the *N*th element of the array *ARRAY*.

Causes the error, Arg not array, if *ARRAY* is not an array. Causes the error, Illegal Arg, if *N* is out of bounds.

(**SETA** *ARRAY N VAL*) [Function]

Sets the *N*th element of *ARRAY* to *VAL*, and returns *VAL*.

Causes the error, Arg not array, if *ARRAY* is not an array. the error, Illegal Arg, if *N* is out of bounds. Can cause the error, Non-numeric arg, if *ARRAY* is an array whose ARRAYTYP is FIXP or FLOATP and *VAL* is non-numeric.

(**ARRAYTYP** *ARRAY*) [Function]

Returns the type of the elements in *ARRAY*, a value corresponding to the second argument to ARRAY.

If ARRAY coerced the array type as described above, ARRAYTYP returns the *new* type. For example, (ARRAYTYP (ARRAY 10 '(BITS 3))) returns BYTE.

(**ARRAYSIZE** *ARRAY*) [Function]

Returns the size of *ARRAY*. Generates the error, Arg not array, if *ARRAY* is not an array.

(**ARRAYORIG** *ARRAY*) [Function]

Returns the origin of *ARRAY*, which may be 0 or 1. Generates an error, Arg not array, if *ARRAY* is not an array.

(**COPYARRAY** *ARRAY*) [Function]

Returns a new array of the same size and type as *ARRAY*, and with the same contents as *ARRAY*. Generates an error, Arg not array, if *ARRAY* is not an array.

[This page intentionally left blank]

6. HASHARRAYS

Hash arrays let you associate arbitrary Lisp objects ("hash keys") with other objects ("hash values"), so you can get from key to value quickly. There are functions for creating hash arrays, putting a hash key/value pair in a hash array, and quickly retrieving the hash value associated with a given hash key.

By default, the hash array functions use `EQ` for comparing hash keys. This means that if non-symbols are used as hash keys, the exact same object (not a copy) must be used to retrieve the hash value. However, you can specify the function used to compare hash keys and to "hash" a hash key to a number. You can, for example, create hash arrays where `EQUAL` but non-`EQ` strings will hash to the same value. Specifying alternative hashing algorithms is described below.

In the description of the functions below, the argument *HARRAY* should be a hasharray created by `HASHARRAY`. For convenience in interactive program development, it may also be `NIL`, in which case a hash array (`SYSHASHARRAY`) provided by the system is used; you must watch out for confusions if this form is used to associate more than one kind of value with the same key.

Note: For backwards compatibility, the hash array functions will accept a list whose `CAR` is a hash array, and whose `CDR` is the "overflow method" for the hash array (see below). However, hash array functions are guaranteed to perform with maximum efficiency only if a direct value of `HASHARRAY` is given.

Note: Interlisp hash arrays and Common Lisp hash tables are the same data type, so functions from both may be intermixed. The only difference between the functions may be argument order, as in `MAPHASH` and `CL:MAPHASH` (see below).

(**HASHARRAY** *MINKEYS OVERFLOW HASHBITSFN EQUIVFN RECLAIMABLE REHASH-THRESHOLD*) [Function]

Creates a hash array with space for at least *MINKEYS* hash keys, with overflow method *OVERFLOW*. See discussion of overflow behavior below.

If *HASHBITSFN* and *EQUIVFN* are non-`NIL`, they specify the hashing function and comparison function used to interpret hash keys. This is described in the section on user-specified hashing functions below. If *HASHBITSFN* and *EQUIVFN* are `NIL`, the default is to hash `EQ` hash keys to the same value.

If *RECLAIMABLE* is *T* the entries in the hash table will be removed if the key has a reference count of one and the table is about to be rehashed. This allows the system, in some cases, to reuse keys instead of expanding the table.

Note: `CL:MAKE-HASH-TABLE` does not allow you to specify your own hashing functions but does provide three built-in types specified by *Common Lisp, the Language.*

(**HARRAY** *MINKEYS*) [Function]

Provided for backward compatibility, this is equivalent to `(HASHARRAY MINKEYS 'ERROR)`, i.e. if the resulting hasarray gets full, an error occurs.

(**HARRAYP** *X*) [Function]

Returns *X* if it is a hash array; otherwise NIL.

HARRAYP returns NIL if *X* is a list whose CAR is an HARRAYP, even though this is accepted by the hash array functions (see below).

(**PUTHASH** *KEY VAL HARRAY*) [Function]

Associates the hash value *VAL* with the hash key *KEY* in *HARRAY*. Replaces the previous hash value, if any. If *VAL* is NIL, any old association is removed (hence a hash value of NIL is not allowed). If *HARRAY* is full when PUTHASH is called with a key not already in the hash array, the function HASHOVERFLOW is called, and the PUTHASH is applied to the value returned (see below). Returns *VAL*.

(**GETHASH** *KEY HARRAY*) [Function]

Returns the hash value associated with the hash key *KEY* in *HARRAY*. Returns NIL, if *KEY* is not found.

(**CLRHASH** *HARRAY*) [Function]

Clears all hash keys/values from *HARRAY*. Returns *HARRAY*.

(**HARRAYPROP** *HARRAY PROP NEWVALUE*) [NoSpread Function]

Returns the property *PROP* of *HARRAY*; *PROP* can have the system-defined values SIZE (the maximum occupancy of *HARRAY*), NUMKEYS (number of occupied slots), OVERFLOW (overflow method), HASHBITSFN (hashing function) and EQUIVFN (comparison function). Except for SIZE and NUMKEYS, a new value may be specified as *NEWVALUE*.

By using other values for *PROP*, the user may also set and get arbitrary property values, to associate additional information with a hash array.

The HASHBITSFN or EQUIVFN properties can only be changed if the hash array is empty.

(**HARRAYSIZE** *HARRAY*) [Function]

Returns the number of slots in *HARRAY*. It's equivalent to (HARRAYPROP HARRAY 'SIZE).

(**REHASH** *OLDHARRAY NEWHARRAY*) [Function]

Hashes all hash keys and values in *OLDHARRAY* into *NEWHARRAY*. The two hash arrays do not have to be (and usually aren't) the same size. Returns *NEWHARRAY*.

(**MAPHASH** *HARRAY MAPHFN*) [Function]

MAPHFN is a function of two arguments. For each hash key in *HARRAY*, *MAPHFN* will be applied to the hash value, and the hash key. For example:

```
[MAPHASH A
   (FUNCTION (LAMBDA (VAL KEY)
       (if (LISTP KEY) then (PRINT VAL)]
```

will print the hash value for all hash keys that are lists. MAPHASH returns *HARRAY*.

Note: the argument order for CL:MAPHASH is *MAPHFN HARRAY*.

(**DMPHASH** $HARRAY_1$ $HARRAY_2$ ... $HARRAY_N$) [NLambda NoSpread Function]

Prints on the primary output file LOADable forms which will restore the hash-arrays contained as the values of the atoms $HARRAY_1$, $HARRAY_2$, ... $HARRAY_N$. Example: (DMPHASH SYSHASHARRAY) will dump the system hash-array.

All EQ identities except symbols and small integers are lost by dumping and loading because READ will create new structure for each item. Thus if two lists contain an EQ substructure, when they are dumped and loaded back in, the corresponding substructures while EQUAL are no longer EQ. The HORRIBLEVARS file package command (Chapter 17) provides a way of dumping hash tables such that these identities are preserved.

Hash Overflow

When a hash array becomes full, trying to add another hash key will cause the function HASHOVERFLOW to be called. This either enlarges the hash array, or causes the error Hash table full. How hash overflow is handled is determined by the value of the OVERFLOW property of the hash array (which can be accessed by HARRAYPROP). The possibilities for the overflow method are:

the symbol ERROR	The error Hash array full is generated when the hash array overflows. This is the default overflow behavior for hash arrays returned by HARRAY.
NIL	The array is automatically enlarged by at least a factor 1.5 every time it overflows. This is the default overflow behavior for hash arrays returned by HASHARRAY.
a positive integer *N*	The array is enlarged to include at least *N* more slots than it currently has.
a floating point number *F*	The array is changed to include *F* times the number of current slots.
a function or lambda expression *FN*	Upon hash overflow, *FN* is called with the hash array as its argument. If *FN* returns a number, that will become the size of the array. Otherwise, the new size defaults to 1.5 times its previous size. *FN* could be used to print a message, or perform some monitor function.

Note: For backwards compatibility, the hash array functions accept a list whose CAR is the hash array, and whose CDR is the overflow method. In this case, the overflow method specified in the list overrides the overflow method set in the hash array. Hash array functions perform with maximum efficiency only if a direct value of HASHARRAY is given.

Specifying Your Own Hashing Functions

In general terms, when a key is looked up in a hash array, it is converted to an integer, which is used to index into a linear array. If the key is not the same as the one found at that index, other indices are tried until it the desired key is found. The value stored with that key is then returned (from GETHASH) or replaced (from PUTHASH).

To customize hash arrays, you'll need to supply the "hashing function" used to convert a key to an integer and the comparison function used to compare the key found in the array with the key being looked up. For hash arrays to work correctly, any two objects which are equal according to the comparison function must "hash" to equal integers.

By default, Medley uses a hashing function that computes an integer from the internal address of a key, and use EQ for comparing keys. This means that if non-atoms are used as hash keys, *the exact same object* (not a copy) must be used to retrieve the hash value.

There are some applications for which the EQ constraint is too restrictive. For example, it may be useful to use strings as hash keys, without the restriction that EQUAL but not EQ strings are considered to be different hash keys.

The user can override this default behavior for any hash array by specifying the functions used to compare keys and to "hash" a key to a number. This can be done by giving the *HASHBITSFN* and *EQUIVFN* arguments to HASHARRAY (see above).

The *EQUIVFN* argument is a function of two arguments that returns non-NIL when its arguments are considered equal. The *HASHBITSFN* argument is a function of one argument that produces a positive small integer (in the range $[0..2^{16} - 1]$) with the property that objects that are considered equal by the *EQUIVFN* produce the same hash bits.

For an existing hash array, the function HARRAYPROP (see above) can be used to examine the hashing and equivalence functions as the HASHBITSFN and EQUIVFN hash array properties. These properties are read-only for non-empty hash arrays, as it makes no sense to change the equivalence relationship once some keys have been hashed.

The following function is useful for creating hash arrays that take strings as hash keys:

(**STRINGHASHBITS** *STRING*) [Function]

Hashes the string *STRING* into an integer that can be used as a HASHBITSFN for a hash array. Strings which are STREQUAL hash to the same integer.

Example:

(HASHARRAY *MINKEYS OVERFLOW* 'STRINGHASHBITS 'STREQUAL)

creates a hash array where you can use strings as hash keys.

[This page intentionally left blank]

7. NUMBERS AND ARITHMETIC FUNCTIONS

There are four different types of numbers in Interlisp: small integers, large integers, bignums (arbitrary-size integers), and floating-point numbers. Small integers are in the range -65536 to 65535. Large integers and floating-point numbers are 32-bit quantities that are stored by “boxing” the number (see below). Bignums are “boxed” as a series of words.

Large integers and floating-point numbers can be any full word quantity. To distinguish among the various kinds of numbers, and other Interlisp pointers, these numbers are “boxed” When a large integer or floating-point number is created (by an arithmetic operation or by `READ`), Interlisp gets a new word from “number storage” and puts the number into that word. Interlisp then passes around the pointer to that word, i.e., the “boxed number”, rather than the actual quantity itself. When a numeric function needs the actual numeric quantity, it performs the extra level of addressing to obtain the “value” of the number. This latter process is called “unboxing”. Unboxing does not use any storage, but each boxing operation uses one new word of number storage. If a computation creates many large integers or floating-point numbers, i.e., does lots of boxes, it may cause a garbage collection of large integer space, or of floating-point number space.

The following functions can be used to distinguish the different types of numbers:

(**SMALLP** *X*) [Function]

Returns *X*, if *X* is a small integer; `NIL` otherwise. Does not generate an error if *X* is not a number.

(**FIXP** *X*) [Function]

Returns *X*, if *X* is an integer; `NIL` otherwise. Note that `FIXP` is true for small integers, large integers, and bignums. Does not generate an error if *X* is not a number.

(**FLOATP** *X*) [Function]

Returns *X* if *X* is a floating-point number; `NIL` otherwise. Does not give an error if *X* is not a number.

(**NUMBERP** *X*) [Function]

Returns *X*, if *X* is a number of any type; `NIL` otherwise. Does not generate an error if *X* is not a number.

Note: In previous releases, `NUMBERP` was true only if (`FLOATP` *X*) or (`FIXP` *X*) were true. With the additon of Common Lisp ratios and complex numbers, `NUMBERP` now returns `T` for *all* number types . Code relying on the "old" behavior should be modified.

Each small integer has a unique representation, so `EQ` may be used to check equality. `EQ` should not be used for large integers, bignums, or floating-point numbers, `EQP`, `IEQP`, or `EQUAL` must be used instead.

(**EQP** *X* *Y*) [Function]

Returns `T`, if *X* and *Y* are equal numbers; `NIL` otherwise. `EQ` may be used if *X* and *Y* are known to be small integers. `EQP` does not convert *X* and *Y* to integers, e.g., `(EQP 2000`

`2000.3) => NIL`, but it can be used to compare an integer and a floating-point number, e.g., `(EQP 2000 2000.0) => T`. `EQP` does not generate an error if *X* or *Y* are not numbers.

`EQP` can also be used to compare stack pointers (see Chapter 11) and compiled code objects (see Chapter 10).

The action taken on division by zero and floating-point overflow is determined with the following function:

(**OVERFLOW** *FLG*) [Function]

Sets a flag that determines the system response to arithmetic overflow (for floating-point arithmetic) and division by zero; returns the previous setting.

For integer arithmetic: If *FLG* = `T`, an error occurs on division by zero. If *FLG* = `NIL` or `0`, integer division by zero returns zero. Integer overflow cannot occur, because small integers are converted to bignums (see the beginning of this chapter).

For floating-point arithmetic: If *FLG* = `T`, an error occurs on floating overflow or floating division by zero. If *FLG* = `NIL` or `0`, the largest (or smallest) floating-point number is returned as the result of the overflowed computation or floating division by zero.

The default value for `OVERFLOW` is `T`, meaning an error is generated on division by zero or floating overflow.

Generic Arithmetic

The functions in this section are “generic” arithmetic functions. If any of the arguments are floating-point numbers (see the Floating-Point Arithmetic section below), they act exactly like floating-point functions, floating all arguments and returning a floating-point number as their value. Otherwise, they act like the integer functions (see the Integer Arithmetic section below). If given a non-numeric argument, they generate an error, `Non-numeric arg`. The results of division by zero and floating-point overflow is determined by the function `OVERFLOW` (see the section above).

(**PLUS** X_1 X_2 ... X_N) [NoSpread Function]

$X_1 + X_2 + \ldots + X_N$.

(**MINUS** *X*) [Function]

$- X$

(**DIFFERENCE** *X* *Y*) [Function]

$X - Y$

(**TIMES** X_1 X_2 ... X_N) [NoSpread Function]

$X_1 * X_2 * \ldots * X_N$

(**QUOTIENT** *X Y*) [Function]

If *X* and *Y* are both integers, returns the integer division of *X* and *Y*. Otherwise, converts both *X* and *Y* to floating-point numbers, and does a floating-point division.

(**REMAINDER** *X Y*) [Function]

If *X* and *Y* are both integers, returns (IREMAINDER *X Y*), otherwise (FREMAINDER *X Y*).

(**GREATERP** *X Y*) [Function]

T, if *X* > *Y*, NIL otherwise.

(**LESSP** *X Y*) [Function]

T if *X* < *Y*, NIL otherwise.

(**GEQ** *X Y*) [Function]

T, if *X* >= *Y*, NIL otherwise.

(**LEQ** *X Y*) [Function]

T, if *X* <= *Y*, NIL otherwise.

(**ZEROP** *X*) [Function]

The same as (EQP *X* 0).

(**MINUSP** *X*) [Function]

T, if *X* is negative; NIL otherwise. Works for both integers and floating-point numbers.

(**MIN** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the minimum of X_1, X_2, . . . , X_N. (MIN) returns the value of MAX.INTEGER (see the Integer Arithmetic section below).

(**MAX** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the maximum of X_1, X_2, ..., X_N. (MAX) returns the value of MIN.INTEGER (see the Integer Arithmetic section below).

(**ABS** *X*) [Function]

X if *X* > 0, otherwise -*X*. ABS uses GREATERP and MINUS (not IGREATERP and IMINUS).

Integer Arithmetic

The input syntax for an integer is an optional sign (+ or -) followed by a sequence of decimal digits, and terminated by a delimiting character. Integers entered with this syntax are interpreted as decimal integers. Integers in other radices can be entered as follows:

123Q
#o123 If an integer is followed by the letter Q, or preceeded by a pound sign and the letter "o", the digits are interpreted as an octal (base 8) integer.

#b10101 If an integer is preceeded by a pound sign and the letter "b", the digits are interpreted as a binary (base 2) integer.

#x1A90 If an integer is preceeded by a pound sign and the letter "x", the digits are interpreted as a hexadecimal (base 16) integer.

#5r1243 If an integer is preceeded by a pound sign, a positive decimal integer BASE, and the letter "r", the digits are interpreted as an integer in the base BASE. For example, #8r123 = 123Q, and #16r12A3 = #x12A3. When typing a number in a radix above ten, the uppercase letters A through Z can be used as the digits after 9 (but there is no digit above Z, so it is not possible to type all base-99 digits).

Medley keeps no record of how you typed a number, so 77Q and 63 both correspond to the same integer, and are indistinguishable internally. The function RADIX (see Chapter 25), sets the radix used to print integers.

PACK and MKATOM create numbers when given a sequence of characters observing the above syntax, e.g. (PACK '(1 2 Q)) => 10. Integers are also created as a result of arithmetic operations.

The range of integers of various types is implementation-dependent. This information is accessible to you through the following variables:

MIN.SMALLP [Variable]
MAX.SMALLP [Variable]

The smallest/largest possible small integer.

MIN.FIXP [Variable]
MAX.FIXP [Variable]

The smallest/largest possible large integer.

MIN.INTEGER [Variable]
MAX.INTEGER [Variable]

The value of MAX.INTEGER and MIN.INTEGER are two special system datatypes. For some algorithms, it is useful to have an integer that is larger than any other integer. Therefore, the values of MAX.INTEGER and MIN.INTEGER are two special data types; the value of MAX.INTEGER is GREATERP than any other integer, and the value of MIN.INTEGER is LESSP than any other integer. Trying to do arithmetic using these special bignums, other than comparison, will cause an error.

All of the functions described below work on integers. Unless specified otherwise, if given a floating-point number, they first convert the number to an integer by truncating the fractional bits, e.g., (IPLUS 2.3 3.8) = 5; if given a non-numeric argument, they generate an error, Non-numeric arg.

(**IPLUS** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the sum $X_1 + X_2 + \ldots + X_N$. (IPLUS) = 0.

(**IMINUS** *X*) [Function]

-*X*

(**IDIFFERENCE** *X Y*) [Function]

$X - Y$

(**ADD1** *X*) [Function]

$X + 1$

(**SUB1** *X*) [Function]

$X - 1$

(**ITIMES** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the product $X_1 * X_2 * \ldots * X_N$. `(ITIMES) = 1`.

(**IQUOTIENT** *X Y*) [Function]

X / *Y* truncated. Examples:

```
(IQUOTIENT 3 2) => 1
(IQUOTIENT -3 2) => -1
```

If *Y* is zero, the result is determined by the function `OVERFLOW` .

(**IREMAINDER** *X Y*) [Function]

Returns the remainder when *X* is divided by *Y*. Example:

```
(IREMAINDER 5 2) => 1
```

(**IMOD** *X N*) [Function]

Computes the integer modulus of *X* mod *N*; this differs from `IREMAINDER` in that the result is always a non-negative integer in the range [0,*N*).

(**IGREATERP** *X Y*) [Function]

`T`, if *X* > *Y*; `NIL` otherwise.

(**ILESSP** *X Y*) [Function]

`T`, if *X* < *Y*; `NIL` otherwise.

(**IGEQ** *X Y*) [Function]

`T`, if *X* >= *Y*; `NIL` otherwise.

(**ILEQ** *X Y*) [Function]

`T`, if *X* <= *Y*; `NIL` otherwise.

(**IMIN** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the minimum of X_1, X_2, . . . , X_N. `(IMIN)` returns the largest possible large integer, the value of `MAX.INTEGER`.

(**IMAX** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the maximum of X_1, X_2, . . . , X_N. (IMAX) returns the smallest possible large integer, the value of MIN.INTEGER.

(**IEQP** *X* *Y*) [Function]

Returns T if *X* and *Y* are equal integers; NIL otherwise. Note that EQ may be used if *X* and *Y* are known to be small integers. IEQP converts *X* and *Y* to integers, e.g., (IEQP 2000 2000.3) => T.

(**FIX** *N*) [Function]

If *N* is an integer, returns *N*. Otherwise, converts *N* to an integer by truncating fractional bits For example, (FIX 2.3) => 2, (FIX -1.7) => -1.

Since FIX is also a programmer's assistant command (see Chapter 13), typing FIX directly to a Medley executive will not cause the function FIX to be called.

(**FIXR** *N*) [Function]

If *N* is an integer, returns *N*. Otherwise, converts *N* to an integer by rounding. FIXR will round towards the even number if *N* is exactly half way between two integers. For example, (FIXR 2.3) => 2, (FIXR -1.7) => -2, (FIXR 3.5) => 4).

(**GCD** N_1 N_2) [Function]

Returns the greatest common divisor of N_1 and N_2, (GCD 72 64)=8.

Logical Arithmetic Functions

(**LOGAND** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the logical AND of all its arguments, as an integer. Example:

```
(LOGAND 7 5 6)  =>  4
```

(**LOGOR** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the logical OR of all its arguments, as an integer. Example:

```
(LOGOR 1 3 9)  =>  11
```

(**LOGXOR** X_1 X_2 . . . X_N) [NoSpread Function]

Returns the logical exclusive OR of its arguments, as an integer. Example:

```
(LOGXOR 11 5)  =>  14
(LOGXOR 11 5 9) = (LOGXOR 14 9)  =>  7
```

(**LSH** *X* *N*) [Function]

(Arithmetic) "Left Shift." Returns *X* shifted left *N* places, with the sign bit unaffected. *X* can be positive or negative. If *N* is negative, *X* is shifted right -*N* places.

(**RSH** *X N*) [Function]

(Arithmetic) "Right Shift." Returns *X* shifted right *N* places, with the sign bit unaffected, and copies of the sign bit shifted into the leftmost bit. *X* can be positive or negative. If *N* is negative, *X* is shifted left -*N* places.

Warning: Be careful if using `RSH` to simulate division; `RSH`ing a negative number isn't the same as dividing by a power of two.

(**LLSH** *X N*) [Function]
(**LRSH** *X N*) [Function]

"Logical Left Shift" and "Logical Right Shift". The difference between a logical and arithmetic right shift lies in the treatment of the sign bit. Logical shifting treats it just like any other bit; arithmetic shifting will not change it, and will "propagate" rightward when actually shifting rightwards. Note that shifting (arithmetic) a negative number "all the way" to the right yields `-1`, not `0`.

Note: `LLSH` and `LRSH` always operate mod-2^{32} arithmetic. Passing a bignum to either of these will cause an error. `LRSH` of negative numbers will shift 0s into the high bits.

(**INTEGERLENGTH** *X*) [Function]

Returns the number of bits needed to represent *X*. This is equivalent to: `1+floor[log2[abs[`*X*`]]]`. `(INTEGERLENGTH 0) = 0`.

(**POWEROFTWOP** *X*) [Function]

Returns non-`NIL` if *X* (coerced to an integer) is a power of two.

(**EVENP** *X Y*) [NoSpread Function]

If *Y* is not given, equivalent to `(ZEROP (IMOD `*X*` 2))`; otherwise equivalent to `(ZEROP (IMOD X Y))`.

(**ODDP** *N MODULUS*) [NoSpread Function]

Equivalent to `(NOT (EVENP N MODULUS))`. *MODULUS* defaults to 2.

(**LOGNOT** *N*) [Macro]

Logical negation of the bits in *N*. Equivalent to `(LOGXOR `*N*` -1)`.

(**BITTEST** *N MASK*) [Macro]

Returns `T` if any of the bits in *MASK* are on in the number *N*. Equivalent to `(NOT (ZEROP (LOGAND `*N MASK*`)))`.

(**BITCLEAR** *N MASK*) [Macro]

Turns off bits from *MASK* in *N*. Equivalent to `(LOGAND `*N*` (LOGNOT `*MASK*`))`.

(**BITSET** *N MASK*) [Macro]

Turns on the bits from *MASK* in *N*. Equivalent to `(LOGOR `*N MASK*`)`.

(**MASK.1'S** *POSITION SIZE*) [Macro]

Returns a bit-mask with *SIZE* one-bits starting with the bit at *POSITION*. Equivalent to (LLSH (SUB1 (EXPT 2 *SIZE*)) *POSITION*).

(**MASK.0'S** *POSITION SIZE*) [Macro]

Returns a bit-mask with all one bits, except for *SIZE* bits starting at *POSITION*. Equivalent to (LOGNOT (MASK.1'S *POSITION SIZE*)).

(**LOADBYTE** *N POS SIZE*) [Function]

Extracts *SIZE* bits from *N*, starting at position *POS*. Equivalent to (LOGAND (RSH *N POS*) (MASK.1'S 0 *SIZE*)).

(**DEPOSITBYTE** *N POS SIZE VAL*) [Function]

Insert *SIZE* bits of *VAL* at position *POS* into *N*, returning the result. Equivalent to

```
(LOGOR (BITCLEAR N (MASK.1'S POS SIZE))
  (LSH (LOGAND VAL (MASK.1'S 0 SIZE))
    POS))
```

(**ROT** *X N FIELDSIZE*) [Function]

"Rotate bits in field". It performs a bitwise left-rotation of the integer *X*, by *N* places, within a field of *FIELDSIZE* bits wide. Bits being shifted out of the position selected by (EXPT 2 (SUB1 *FIELDSIZE*)) will flow into the "units" position.

The notions of position and size can be combined to make up a "byte specifier", which is constructed by the macro BYTE [note reversal of arguments as compared with the above functions]:

(**BYTE** *SIZE POSITION*) [Macro]

Constructs and returns a "byte specifier" containing *SIZE* and *POSITION*.

(**BYTESIZE** *BYTESPEC*) [Macro]

Returns the SIZE componant of the "byte specifier" *BYTESPEC*.

(**BYTEPOSITION** *BYTESPEC*) [Macro]

Returns the POSITION componant of the "byte specifier" *BYTESPEC*.

(**LDB** *BYTESPEC VAL*) [Macro]

Equivalent to

(LOADBYTE *VAL* (BYTEPOSITION *BYTESPEC*) (BYTESIZE *BYTESPEC*))

(**DPB** *N BYTESPEC VAL*) [Macro]

Equivalent to

(DEPOSITBYTE *VAL* (BYTEPOSITION *BYTESPEC*) (BYTESIZE *BYTESPEC*) *N*)

Floating-Point Arithmetic

A floating-point number is input as a signed integer, followed by a decimal point, and another sequence of digits called the fraction, followed by an exponent (represented by `E` followed by a signed integer) and terminated by a delimiter.

Both signs are optional, and either the fraction following the decimal point, or the integer preceding the decimal point may be omitted. One or the other of the decimal point or exponent may also be omitted, but at least one of them must be present to distinguish a floating-point number from an integer. For example, the following will be recognized as floating-point numbers:

```
5.      5.00    5.01    .3
5E2     5.1E2   5E-3    -5.2E+6
```

Floating-point numbers are printed using the format control specified by the function `FLTFMT` (see Chapter 25). `FLTFMT` is initialized to `T`, or free format. For example, the above floating-point numbers would be printed free format as:

```
5.0     5.0     5.01    .3
500.0   510.0   .005    -5.2E6
```

Floating-point numbers are created by the reader when a "`.`" or an `E` appears in a number, e.g., `1000` is an integer, `1000.` a floating-point number, as are `1E3` and `1.E3`. Note that `1000D`, `1000F`, and `1E3D` are perfectly legal literal atoms. Floating-point numbers are also created by `PACK` and `MKATOM`, and as a result of arithmetic operations.

`PRINTNUM` (see Chapter 25) permits greater control over the printed appearance of floating-point numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

The floating-point number range is stored in the following variables:

MIN.FLOAT [Variable]

The smallest possible floating-point number.

MAX.FLOAT [Variable]

The largest possible floating-point number.

All of the functions described below work on floating-point numbers. Unless specified otherwise, if given an integer, they first convert the number to a floating-point number, e.g., `(FPLUS 1 2.3)` `<=> (FPLUS 1.0 2.3) => 3.3`; if given a non-numeric argument, they generate an error, `Non-numeric arg`.

(**FPLUS** X_1 X_2 . . . X_N) [NoSpread Function]

$X_1 + X_2 + \ldots + X_N$

(**FMINUS** *X*) [Function]

$- X$

(**FDIFFERENCE** *X* *Y*) [Function]

$X - Y$

(**FTIMES** X_1 X_2 ... X_N) [NoSpread Function]

X_1 * X_2 * ... * X_N

(**FQUOTIENT** *X Y*) [Function]

X / *Y*.

The results of division by zero and floating-point overflow is determined by the function `OVERFLOW`.

(**FREMAINDER** *X Y*) [Function]

Returns the remainder when *X* is divided by *Y*. Equivalent to:

```
(FDIFFERENCE X (FTIMES Y (FIX (FQUOTIENT X Y))))
```

Example:

```
(FREMAINDER 7.5 2.3) => 0.6
```

(**FGREATERP** *X Y*) [Function]

`T`, if *X* > *Y*, `NIL` otherwise.

(**FLESSP** *X Y*) [Function]

`T`, if *X* < *Y*, `NIL` otherwise.

(**FEQP** *X Y*) [Function]

Returns `T` if *X* and *Y* are equal floating-point numbers; `NIL` otherwise. `FEQP` converts *X* and *Y* to floating-point numbers.

(**FMIN** X_1 X_2 ... X_N) [NoSpread Function]

Returns the minimum of X_1, X_2, ..., X_N. `(FMIN)` returns the largest possible floating-point number, the value of `MAX.FLOAT`.

(**FMAX** X_1 X_2 ... X_N) [NoSpread Function]

Returns the maximum of X_1, X_2, ..., X_N. `(FMAX)` returns the smallest possible floating-point number, the value of `MIN.FLOAT`.

(**FLOAT** *X*) [Function]

Converts *X* to a floating-point number. Example:

```
(FLOAT 0) => 0.0
```

Transcendental Arithmetic Functions

(**EXPT** *A N*) [Function]

Returns A^N. If *A* is an integer and *N* is a positive integer, returns an integer, e.g, `(EXPT 3 4) => 81`, otherwise returns a floating-point number. If *A* is negative and *N* fractional, generates the error, `Illegal exponentiation`. If *N* is floating and either too large or too small, generates the error, `Value out of range expt`.

(**SQRT** *N*) [Function]

Returns the square root of *N* as a floating-point number. *N* may be fixed or floating-point. Generates an error if *N* is negative.

(**LOG** *X*) [Function]

Returns the natural logarithm of *X* as a floating-point number. *X* can be integer or floating-point.

(**ANTILOG** *X*) [Function]

Returns the floating-point number whose logarithm is *X*. *X* can be integer or floating-point. Example:

```
(ANTILOG 1) = e => 2.71828...
```

(**SIN** *X RADIANSFLG*) [Function]

Returns the sine of *X* as a floating-point number. *X* is in degrees unless *RADIANSFLG* = `T`.

(**COS** *X RADIANSFLG*) [Function]

Similar to `SIN`.

(**TAN** *X RADIANSFLG*) [Function]

Similar to `SIN`.

(**ARCSIN** *X RADIANSFLG*) [Function]

The value of `ARCSIN` is a floating-point number, and is in degrees unless *RADIANSFLG* = `T`. In other words, if `(ARCSIN` *X RADIANSFLG*`) = Z` then `(SIN Z` *RADIANSFLG*`) =` *X*. The range of the value of `ARCSIN` is -90 to +90 for degrees, $-\pi/2$ to $\pi/2$ for radians. *X* must be a number between -1 and 1.

(**ARCCOS** *X RADIANSFLG*) [Function]

Similar to `ARCSIN`. Range is `0` to `180`, 0 to π.

(**ARCTAN** *X RADIANSFLG*) [Function]

Similar to `ARCSIN`. Range is `0` to `180`, 0 to π.

(**ARCTAN2** *Y X RADIANSFLG*) [Function]

Computes `(ARCTAN (FQUOTIENT` *Y X*`)` *RADIANSFLG*`)`, and returns a corresponding value in the range `-180` to `180` (or $-\pi$ to π), i.e. the result is in the proper quadrant as determined by the signs of *X* and *Y*.

Generating Random Numbers

(**RAND** *LOWER UPPER*) [Function]

Returns a pseudo-random number between *LOWER* and *UPPER* inclusive, i.e., RAND can be used to generate a sequence of random numbers. If both limits are integers, the value of RAND is an integer, otherwise it is a floating-point number. The algorithm is completely deterministic, i.e., given the same initial state, RAND produces the same sequence of values. The internal state of RAND is initialized using the function RANDSET.

(**RANDSET** *X*) [Function]

Returns the internal state of RAND. If *X* = NIL, just returns the current state. If *X* = T, RAND is initialized using the clocks, and RANDSET returns the new state. Otherwise, *X* is interpreted as a previous internal state, i.e., a value of RANDSET, and is used to reset RAND. For example,

```
←(SETQ OLDSTATE (RANDSET))
 ...
←(for X from 1 to 10 do (PRIN1 (RAND 1 10)))
 2847592748NIL
←(RANDSET OLDSTATE)
 ...
←(for X from 1 to 10 do (PRIN1 (RAND 1 10)))
 2847592748NIL
```

[This page intentionally left blank]

Hiding the details of your code makes it more readable, and lets you program more efficiently. Data structures are a good example: You're better off if you can say "Fetch me the SPEED field from this AIRPLANE" rather than having to say (CAR (CDDDR (CADR AIRPLANE))). You can declare data structures used by your programs, then work with field names rather than access details. Using the declarations, Medley performs the access/storage operations you request. If you change a data structure's declaration, your programs automatically adjust.

You describe the format of a data structure (record) by making a "record declaration" (see the Record Declarations section below). The record declaration is a description of the record, associating names with its various parts, or "fields". For example, the record declaration

```
(RECORD MSG  (FROM TO TEXT))
```

describes a data structure called MSG, that has three fields: FROM, TO, and TEXT. You can refer to these fields by name, to get their values or to store new values into them, by using FETCH and REPLACE:

```
(fetch (MSG FROM)of MYMSG)
(replace (MSG TO) of MYMSG with "John Doe")
```

You create new MSGs with CREATE:

```
(SETQ MYMSG (create MSG))
```

and TYPE? tells you whether some object is a MSG:

```
(IF (TYPE? MSG THIS-THING) then (SEND-MSG THIS-THING))
```

So far we've said nothing about *how* your MSG is represented—when you're writing FETCHes and REPLACEs, it doesn't matter. But you *can* control the representation: The symbol RECORD in the declaration above causes each MSG to be represented as a list. There are a number of options, up to creating a completely new Lisp data type; each has its own specifier symbol, and they're described in detail below.

The record package is implemented using DWIM and CLISP, so it will do spelling correction on field names, record types, etc. Record operations are translated using all CLISP declarations in effect (standard/fast/undoable).

The file manager's RECORDS command lets you give record declarations (see Chapter 17), and FILES? and CLEANUP will tell you about record declarations that need to be dumped.

FETCH and REPLACE

The fields of a record are accessed and changed with FETCH and REPLACE. If X is a MSG data structure, (fetch FROM of X) will return the value of the FROM field of X, and (replace FROM of X with Y) will replace this field with the value of Y. In general, the value of a REPLACE operation is the same as the value stored into the field.

Note that (fetch FROM of X) assumes that X is an instance of the record MSG—the interpretation of (fetch FROM of X) never depends on the *value* of X. If X is not a MSG, this may produce incorrect results.

If there is another record declaration, (RECORD REPLY (TEXT RESPONSE)), then (fetch TEXT of X) is ambiguous, because X could be either a MSG or a REPLY record. In this case, an error will occur, Ambiguous record field. To clarify this, give FETCH and REPLACE a list for their "field" argument: (fetch (MSG TEXT) of X) will fetch the TEXT field of a MSG record. If a field has an *identical* interpretation in two declarations, e.g., if the field

TEXT occurred in the same location within the declarations of MSG and REPLY, then (fetch TEXT of X) would *not* be ambiguous.

If there's a conflict, "user" record declarations take precedence over "system" record declarations. System records are declared by including (SYSTEM) in the declaration (see the Record Declarations section below). All of the records defined in the standard Medley system are system records.

Another complication can occur if the fields of a record are themselves records. The fields of a record can be further broken down into sub-fields by a "subdeclaration" within the record declaration. For example,

```
(RECORD NODE (POSITION . LABEL) (RECORD POSITION (XLOC . YLOC)))
```

lets you access the POSITION field with (fetch POSITION of X), or its subfield XLOC with (fetch XLOC of X).

You may also declare that field name in a *separate* record declaration. For instance, the TEXT field in the MSG and REPLY records above may be subdivided with the seperate record declaration (RECORD TEXT (HEADER TXT)). You get to fields of subfields (to any level of nesting) by specifying the "data path" as a list of record / field names, where there is some path from each record to the next in the list. For instance,

```
(fetch (MSG TEXT HEADER) of X)
```

treats X as a MSG record, fetches its TEXT field, and fetches *its* HEADER field. You only need to give enough of the data path to disambiguate it. In this case, (fetch (MSG HEADER) of X) is sufficient: Medley searches among all current record declarations for a path from each name to the next, considering first local declarations (see Chapter 21) and then global ones. Of course, if you had two records with HEADER fields, you get an Ambiguous data path error.

FETCH and REPLACE are translated using the CLISP declarations in effect (see Chapter 21). FFETCH and FREPLACE are fast versions that don't do any type checking. /REPLACE insures undoable declarations.

Record Declarations

You define records by evaluating declarations of the form:

(*RECORD-TYPE RECORD-NAME RECORD-FIELDS* . *RECORD-TAIL*)

RECORD-TYPE specifies the "type" of data you're declaring, and controls how instances will be stored internally. The different record types are described below.

RECORD-NAME is a symbol used to identify the record declaration for CREATE, TYPE?, FETCH and REPLACE, and dumping to files (see Chapter 17). DATATYPE and TYPERECORD declarations also use *RECORD-NAME* to identify the data structure (as described below).

RECORD-FIELDS describes the structure of the record. Its exact interpretation varies with RECORD-TYPE. Generally, it names the fields within the record that can be accessed with FETCH and REPLACE.

RECORD-TAIL is an optional list where you can specify default values for record fields, special CREATE and TYPE? forms, and subdeclarations (described below).

Record declarations are Lisp programs, and could be included in functions, changing a record declaration at run-time. *Don't do it.* You risk creating a structure with one declaration, and trying to fetch from it with another—complete chaos results. If you need to change record declarations dynamically, consider using association lists or property lists.

Record Types

The *RECORD-TYPE* field of the record declaration specifies how the data object is created, and how the various record fields are accessed. Depending on the record type, the record fields may be stored in a list, or in an array, or on a symbol's property list. The following record types are defined:

RECORD [Record Type]

The fields of a RECORD are kept in a list. *RECORD-FIELDS* is a list; each non-NIL symbol is a field-name to be associated with the corresponding element or tail of a list structure. For example, with the declaration `(RECORD MSG (FROM TO . TEXT))`, `(fetch FROM of X)` translates as `(CAR X)`.

NIL can be used as a place marker for an unnamed field, e.g., `(A NIL B)` describes a three element list, with B corresponding to the third element. A number may be used to indicate a sequence of NILs, e.g. `(A 4 B)` is interpreted as `(A NIL NIL NIL NIL B)`.

DATATYPE [Record Type]

Defines a new user data type with type name *RECORD-NAME*. Unlike other record types, the instances of a DATATYPE are represented with a completely new Lisp type, and not in terms of other existing types.

RECORD-FIELDS is a list of field specifications, where each specification is either a list (*FIELDNAME FIELDTYPE*), or an symbol *FIELDNAME*. If *FIELDTYPE* is omitted, it defaults to POINTER. Possible values for *FIELDTYPE* are:

POINTER Field contains a pointer to any arbitrary Interlisp object.

INTEGER
FIXP Field contains a signed integer. Caution: An INTEGER field is not capable of holding everything that satisfies FIXP, such as bignums.

FLOATING
FLOATP Field contains a floating point number.

SIGNEDWORD Field contains a 16-bit signed integer.

FLAG Field is a one bit field that "contains" T or NIL.

BITS *N* Field contains an *N*-bit unsigned integer.

BYTE Equivalent to BITS 8.

WORD Equivalent to BITS 16.

XPOINTER Field contains a pointer like POINTER, but the field is *not* reference counted by the garbage collector. XPOINTER fields are useful for implementing back-pointers in structures that would be circular and not otherwise collected by the reference-counting garbage collector.

Warning: Use XPOINTER fields with great care. You can damage the integrity of the storage allocation system by using pointers to objects that have been garbage collected. Code that uses XPOINTER fields should be sure that the objects pointed to have not been garbage collected. This can be done in two ways: The first is to maintain the object in a global structure, so that it is never garbage collected until explicitly deleted from the structure, at which point the program must invalidate all the XPOINTER fields of other objects pointing at it. The second is to declare the object as a DATATYPE beginning with a POINTER field that the program maintains as a pointer to an object of another type (e.g., the object containing the XPOINTER pointing back at it), and test that field for reasonableness whenever using the contents of the XPOINTER field.

For example, the declaration

```
(DATATYPE FOO
     ((FLG BITS 12) TEXT HEAD (DATE BITS 18)
     (PRIO FLOATP) (READ? FLAG)))
```

would define a data type FOO with two pointer fields, a floating point number, and fields for a 12 and 18 bit unsigned integers, and a flag (one bit). Fields are allocated in such a way as to optimize the storage used and not necessarily in the order specified. Generally, a DATATYPE record is much more storage compact than the corresponding RECORD structure would be; in addition, access is faster.

Since the user data type must be set up at *run*-time, the RECORDS file package command will dump a DECLAREDATATYPE expression as well as the DATATYPE declaration itself. If the record declaration is otherwise not needed at runtime, it can be kept out of the compiled file by using a (DECLARE: DONTCOPY --) expression (see Chapter 17), but it is still necessary to ensure that the datatype is properly initialized. For this, one can use the INITRECORDS file package command (see Chapter 17), which will dump only the DECLAREDATATYPE expression.

Note: When defining a new data type, it is sometimes useful to call the function DEFPRINT (see Chapter 25) to specify how instances of the new data type should be printed. This can be specified in the record declaration by including an INIT record specification (see the Optional Record Specifications section below), e.g. (DATATYPE QV.TYPE ... (INIT (DEFPRINT 'QV.TYPE (FUNCTION PRINT.QV.TYPE)))).

DATATYPE declarations cannot be used within local record declarations (see Chapter 21).

TYPERECORD [Record Type]

Similar to RECORD, but the record name is added to the front of the list structure to signify what "type" of record it is. This type field is used in the translation of TYPE? expressions. CREATE will insert an extra field containing *RECORD-NAME* at the beginning of the structure, and the translation of the access and storage functions will take this extra field into account. For example, for (TYPERECORD MSG (FROM TO . TEXT)), (fetch FROM of X) translates as (CADR X), not (CAR X).

ASSOCRECORD [Record Type]

Describes lists where the fields are stored in association list format:

((*FIELDNAME*$_1$. *VALUE*$_1$) (*FIELDNAME*$_2$. *VALUE*$_2$) ...)

RECORD-FIELDS is a list of symbols, the permissable field names in the association list. Access is done with ASSOC (or FASSOC, if the current CLISP declarations are FAST, see Chapter 21), storing with PUTASSOC.

PROPRECORD [Record Type]

Describes lists where the fields are stored in property list format:

(*FIELDNAME*$_1$ *VALUE*$_1$ *FIELDNAME*$_2$ *VALUE*$_2$ ...)

RECORD-FIELDS is a list of symbols, the permissable field names in the property list. Access is done with LISTGET, storing with LISTPUT.

Both ASSOCRECORD and PROPRECORD are useful for defining data structures where many of the fields are NIL. CREATEing one these record types only stores those fields that are non-NIL. Note, however, that with the record declaration (PROPRECORD FIE (H I J)) the expression (create FIE) would still construct (H NIL), since a later operation of (replace J of X with Y) could not possibly change the instance of the record if it were NIL.

ARRAYRECORD [Record Type]

ARRAYRECORDs are stored as arrays. *RECORD-FIELDS* is a list of field names that are associated with the corresponding elements of an array. NIL can be used as a place marker for an unnamed field (element). Positive integers can be used as abbreviation for the corresponding number of NILs. For example, (ARRAYRECORD (ORG DEST NIL ID 3 TEXT)) describes an eight-element array, with ORG corresponding to the first element, ID to the fourth, and TEXT to the eighth.

ARRAYRECORD only creates arrays of pointers. Other kinds of arrays must be implemented with ACCESSFNS (see below).

HASHLINK [Record Type]

The HASHLINK record type can be used with any type of data object: it specifies that the value of a single field can be accessed by hashing the data object in a given hash array. Since the HASHLINK record type describes an access method, rather than a data structure, CREATE is meaningless for HASHLINK records.

RECORD-FIELDS is either a symbol *FIELD-NAME*, or a list (*FIELD-NAME HARRAYNAME HARRAYSIZE*). *HARRAYNAME* is a variable whose value is the hash array to be used; if not given, SYSHASHARRAY is used. If the value of the variable *HARRAYNAME* is not a hash array (at the time of the record declaration), it will be set to a new hash array with a size of *HARRAYSIZE*. *HARRAYSIZE* defaults to 100.

The HASHLINK record type is useful as a subdeclaration to other records to add additional fields to already existing data structures (see the Optional Record Specifications section below). For example, suppose that FOO is a record declared with (RECORD FOO (A B C)). To add a new field BAR, without modifying the existing data strutures, redeclare FOO with:

```
(RECORD FOO (A B C) (HASHLINK FOO (BAR BARHARRAY)))
```

Now, (fetch BAR of X) will translate into (GETHASH X BARHARRAY), hashing off the existing list X.

ATOMRECORD [Record Type]

ATOMRECORDs are stored on the property lists of symbols. *RECORD-FIELDS* is a list of property names. Accessing is performed with GETPROP, storing with PUTPROP. The CREATE expression is not initially defined for ATOMRECORD records.

BLOCKRECORD [Record Type]

BLOCKRECORD is used in low-level system programming to "overlay" an organized structure over an arbitrary piece of raw storage. *RECORD-FIELDS* is interpreted exactly as with a DATATYPE declaration, except that fields are *not* automatically rearranged to maximize storage efficiency. Like an ACCESSFNS record, a BLOCKRECORD does not have concrete instances; it merely provides a way of interpreting some existing block of storage. So you can't create an instance of a BLOCKRECORD (unless the declaration includes an explicit CREATE expression), nor is there a default type? expression for a BLOCKRECORD.

Warning: Exercise caution in using BLOCKRECORD declarations, as they let you fetch and store arbitrary data in arbitrary locations, thereby evading Medley's normal type system. Except in very specialized situations, a BLOCKRECORD should never contain POINTER or XPOINTER fields, nor be used to overlay an area of storage that contains pointers. Such use could compromise the garbage collector and storage allocation system. You are responsible for ensuring that all FETCH and REPLACE expressions are performed only on suitable objects, as no type testing is performed.

A typical use for a BLOCKRECORD in user code is to overlay a non-pointer portion of an existing DATATYPE. For this use, the LOCF macro is useful. (LOCF (fetch *FIELD* of *DATUM*)) can be used to refer to the storage that begins at the first word that contains *FIELD* of *DATUM*. For example, to define a new kind of Ethernet packet, you could overlay the "body" portion of the ETHERPACKET datatype declaration as follows:

```
(ACCESSFNS MYPACKET
((MYBASE (LOCF (fetch (ETHERPACKET EPBODY) of DATUM))))
 (BLOCKRECORD MYBASE
 ((MYTYPE WORD) (MYLENGTH WORD) (MYSTATUS BYTE)
  (MYERRORCODE BYTE) (MYDATA INTEGER)))
  (TYPE? (type? ETHERPACKET DATUM)))
```

With this declaration in effect, the expression (fetch MYLENGTH of PACKET) would retrieve the second 16-bit field beyond the place inside PACKET where the EPBODY field starts.

ACCESSFNS [Record Type]

ACCESSFNS lets you specify arbitrary functions to fetch and store data. For each field name, you specify how it is to be accessed and set. This lets you use arbitrary data structures, with complex access methods. Most often, ACCESSFNS are useful when you can compute one field's value from other fields. If you're representing a time period by its start and duration, you could add an ACCESSFNS definition for the ending time that did the obvious addition.

RECORD-FIELDS is a list of elements of the form (*FIELD-NAME ACCESSDEF SETDEF*). *ACCESSDEF* should be a function of one argument, the datum, and will be used for accessing the value of the field. *SETDEF* should be a function of two arguments, the datum and the new value, and will be used for storing a new value in a field. *SETDEF* may be omitted, in which case, no storing operations are allowed.

ACCESSDEF and/or *SETDEF* may also be a form written in terms of variables DATUM and (in *SETDEF*) NEWVALUE. For example, given the declaration

```
[ACCESSFNS FOO
((FIRSTCHAR (NTHCHAR DATUM 1) (RPLSTRING DATUM 1 NEWVALUE)) (RESTCHARS (SUBSTRING DATUM 2]
```

(replace (FOO FIRSTCHAR) of X with Y) would translate to (RPLSTRING X 1 Y). Since no *SETDEF* is given for the RESTCHARS field, attempting to perform (replace (FOO RESTCHARS) of X with Y) would generate an error, Replace undefined for field. Note that ACCESSFNS do not have a CREATE definition. However, you may supply one in the defaults or subdeclarations of the declaration, as described below. Attempting to CREATE an ACCESSFNS record without specifying a create definition will cause an error Create not defined for this record.

ACCESSDEF and *SETDEF* can also be a property list which specify FAST, STANDARD and UNDOABLE versions of the ACCESSFNS forms, e.g.

```
[ACCESSFNS LITATOM
      ((DEF (STANDARD GETD FAST FGETD)
            (STANDARD PUTD UNDOABLE /PUTD]
```

means if FAST declaration is in effect, use FGETD for fetching, if UNDOABLE, use /PUTD for saving (see CLISP declarations, see Chapter 21).

SETDEF forms should be written so that they return the new value, to be consistant with REPLACE operations for other record types. The REPLACE does not enforce this, though.

ACCESSFNS let you use data structures not specified by one of the built-in record types. For example, one possible representation of a data structure is to store the fields in *parallel* arrays, especially if the number of instances required is known, and they needn't be garbage collected. To implement LINK with two fields FROM and TO, you'd have two arrays FROMARRAY and TOARRAY. The representation of an "instance" of LINK would be an integer, used to index into the arrays. This can be accomplished with the declaration:

```
[ACCESSFNS LINK
   ((FROM (ELT FROMARRAY DATUM)
          (SETA FROMARRAY DATUM NEWVALUE))
    (TO (ELT TOARRAY DATUM)
        (SETA TOARRAY DATUM NEWVALUE)))
    (CREATE (PROG1 (SETQ LINKCNT (ADD1 LINKCNT))
                   (SETA FROMARRAY LINKCNT FROM)
                   (SETA TOARRAY LINKCNT TO)))
    (INIT (PROGN
             (SETQ FROMARRAY (ARRAY 100))
             (SETQ TOARRAY (ARRAY 100))
             (SETQ LINKCNT 0)]
```

To create a new LINK, a counter is incremented and the new elements stored. (Note: The CREATE form given the declaration probably should include a test for overflow.)

Optional Record Specifications

After the *RECORD-FIELDS* item in a record declaration expression there can be an arbitrary number of additional expressions in *RECORD-TAIL*. These expressions can be used to specify default values for record fields, special CREATE and TYPE? forms, and subdeclarations. The following expressions are permitted:

FIELD-NAME ← *FORM* Allows you to specify within the record declaration the default value to be stored in *FIELD-NAME* by a CREATE (if no value is given within the CREATE expression itself). Note that *FORM* is evaluated at CREATE time, not when the declaration is made.

(CREATE *FORM*) Defines the manner in which CREATE of this record should be performed. This provides a way of specifying how ACCESSFNS should be created or overriding the usual definition of CREATE. If *FORM* contains the field-names of the declaration as variables, the forms given in the

CREATE operation will be substituted in. If the word DATUM appears in the create form, the *original* CREATE definition is inserted. This effectively allows you to "advise" the create.

(INIT *FORM*) Specifies that *FORM* should be evaluated when the record is declared. *FORM* will also be dumped by the INITRECORDS file package command (see Chapter 17).

For example, see the example of an ACCESSFNS record declaration above. In this example, FROMARRAY and TOARRAY are initialized with an INIT form.

(TYPE? *FORM*) Defines the manner in which TYPE? expressions are to be translated. *FORM* may either be an expression in terms of DATUM or a function of one argument.

(SUBRECORD *NAME* . *DEFAULTS*) *NAME* must be a field that appears in the current declaration and the name of another record. This says that, for the purposes of translating CREATE expressions, substitute the top-level declaration of *NAME* for the SUBRECORD form, adding on any defaults specified.

For example: Given (RECORD B (E F G)), (RECORD A (B C D) (SUBRECORD B)) would be treated like (RECORD A (B C D) (RECORD B (E F G))) for the purposes of translating CREATE expressions.

a subdeclaration If a record declaration expression occurs among the record specifications of another record declaration, it is known as a "subdeclaration." Subdeclarations are used to declare that fields of a record are to be interpreted as another type of record, or that the record data object is to be interpreted in more than one way.

The *RECORD-NAME* of a subdeclaration must be either the *RECORD-NAME* of its immediately superior declaration or one of the superior's field-names. Instead of identifying the declaration as with top level declarations, the record-name of a subdeclaration identifies the parent field or record that is being described by the subdeclaration. Subdeclarations can be nested to an arbitrary depth.

Giving a subdeclaration (RECORD $NAME_1$ $NAME_2$) is a simple way of defining a *synonym* for the field $NAME_1$.

It is possible for a given field to have more than one subdeclaration. For example, in

```
(RECORD FOO (A B) (RECORD A (C D)) (RECORD A (Q R)))
```

(Q R) and (C D) are "overlayed," i.e. (fetch Q of X) and (fetch C of X) would be equivalent. In such cases, the *first* subdeclaration is the one used by CREATE.

(SYNONYM *FIELD*

(*SYN*$_1$ ... *SYN*$_N$)) *FIELD* must be a field that appears in the current declaration. This defines *SYN*$_1$ ... *SYN*$_N$ all as synonyms of *FIELD*. If there is only one synonym, this can be written as (SYNONYM *FIELD SYN*).

(SYSTEM) If (SYSTEM) is included in a record declaration, this indicates that the record is a "system" record rather than a "user" record. The only distinction between the two types of records is that "user" record declarations take precedence over "system" record declarations, in cases where an unqualified field name would be considered ambiguous. All of the records defined in the standard Medley system are defined as system records.

CREATE

You can create RECORDs by hand if you like, using CONS, LIST, etc. But that defeats the whole point of hiding implementation details. So much easier to use:

(create *RECORD-NAME* . *ASSIGNMENTS*)

CREATE translates into an appropriate Interlisp form that uses CONS, LIST, PUTHASH, ARRAY, etc., to create the new datum with the its fields initialized to the values you specify. *ASSIGNMENTS* is optional and may contain expressions of the following form:

FIELD-NAME ← *FORM* Specifies initial value for *FIELD-NAME*.

USING *FORM* *FORM* is an existing instance of *RECORD-NAME*. If you don't specify a value for some field, the value of the corresponding field in *FORM* is to be used.

COPYING *FORM* Like USING, but the corresponding values are copied (with COPYALL).

REUSING *FORM* Like USING, but wherever possible, the corresponding *structure* in *FORM* is used.

SMASHING *FORM* A new instance of the record is not created at all; rather, new field values are smashed into *FORM*, which CREATE then returns.

When it makes a difference, Medley goes to great pains to make its translation do things in the same order as the original CREATE expression. For example, given the declaration (RECORD CONS (CAR . CDR)), the expression (create CONS CDR←X CAR←Y) will translate to (CONS Y X), but (create CONS CDR←(FOO) CAR←(FIE)) will translate to ((LAMBDA ($$1) (CONS (PROGN (SETQ $$1 (FOO)) (FIE)) $$1))) because FOO might set some variables used by FIE.

How are USING and REUSING different? (create *RECORD* reusing *FORM* ...) doesn't do any destructive operations on the value of *FORM*, but *will* incorporate as much as possible of the old data structure into the new one. On the other hand, (create *RECORD* using *FORM* ...) will create a completely new data structure, with only the *contents* of the fields re-used. For example, REUSING a PROPRECORD just CONSes the new property names and values onto the list, while USING copies the top level of the list. Another example of this distinction occurs when a field is elaborated by a subdeclaration: USING will create a new instance of the sub-record, while REUSING will use the old contents of the field (unless some field of the subdeclaration is assigned in the CREATE expression.)

If the value of a field is neither explicitly specified, nor implicitly specified via USING, COPYING or REUSING, the default value in the declaration is used, if any, otherwise NIL. (For BETWEEN fields in DATATYPE records, N_1 is used; for other non-pointer fields zero is used.) For example, following (RECORD A (B C D) D ← 3)

```
(create A B ← T) ==> (LIST T NIL 3)
(create A B ← T using X) ==> (LIST T (CADR X) (CADDR X))
(create A B ← T copying X)) ==> [LIST T (COPYALL (CADR X)) (COPYALL (CADDR X]
(create A B ← T reusing X) ==> (CONS T (CDR X))
```

TYPE?

The record package allows you to test if a given datum "looks like" an instance of a record. This can be done via an expression of the form (type? *RECORD-NAME FORM*).

TYPE? is mainly intended for records with a record type of DATATYPE or TYPERECORD. For DATATYPEs, the TYPE? check is exact; i.e. the TYPE? expression will return non-NIL only if the value of *FORM* is an instance of the record named by *RECORD-NAME*. For TYPERECORDs, the TYPE? expression will check that the value of *FORM* is a list beginning with *RECORD-NAME*. For ARRAYRECORDs, it checks that the value is an array of the correct size. For PROPRECORDs and ASSOCRECORDs, a TYPE? expression will make sure that the value of *FORM* is a property/association list with property names among the field-names of the declaration.

There is no built-in type test for records of type ACCESSFNS, HASHLINK or RECORD. Type tests can be defined for these kinds of records, or redefined for the other kinds, by including an expression of the form (TYPE? *COM*) in the record declaration (see the Record Declarations section below). Attempting to execute a TYPE? expression for a record that has no type test causes an error, Type? not implemented for this record.

WITH

Often one wants to write a complex expression that manipulates several fields of a single record. The WITH construct can make it easier to write such expressions by allowing one to refer to the fields of a record as if they were variables within a lexical scope:

(with *RECORD-NAME RECORD-INSTANCE* $FORM_1$ ... $FORM_N$)

RECORD-NAME is the name of a record, and *RECORD-INSTANCE* is an expression which evaluates to an instance of that record. The expressions $FORM_1$ ... $FORM_N$ are evaluated so that references to variables which are field-names of *RECORD-NAME* are implemented via FETCH and SETQs of those variables are implemented via REPLACE.

For example, given

```
(RECORD RECN (FLD1 FLD2))
(SETQ INST (create RECN FLD1 ← 10 FLD2 ← 20))
```

Then the construct

```
(with RECN INST (SETQ FLD2 (PLUS FLD1 FLD2]
```

is equivalent to

```
(replace FLD2 of INST with (PLUS (fetch FLD1 of INST) (fetch FLD2 of INST]
```

Warning: WITH is implemented by doing simple substitutions in the body of the forms, without regard for how the record fields are used. This means, for example, if the record FOO is defined by (RECORD FOO (POINTER1 POINTER2)), then the form

```
(with FOO X (SELECTQ Y (POINTER1 POINTER1) NIL]
```

will be translated as

```
(SELECTQ Y ((CAR X) (CAR X)) NIL]
```

Be careful that record field names are not used except as variables in the WITH forms.

Defining New Record Types

In addition to the built-in record types, you can declare your own record types by performing the following steps:

1. Add the new record-type to the value of CLISPRECORDTYPES.
2. Perform (MOVD 'RECORD *RECORD-TYPE*).
3. Put the name of a function which will return the translation on the property list of *RECORD-TYPE*, as the value of the property USERRECORDTYPE. Whenever a record declaration of type *RECORD-TYPE* is encountered, this function will be passed the record declaration as its argument, and should return a *new* record declaration which the record package will then use in its place.

Manipulating Record Declarations

(**EDITREC** *NAME* COM_1 ... COM_N) [NLambda NoSpread Function]

EDITREC calls the editor on a copy of all declarations in which *NAME* is the record name or a field name. On exit, it redeclares those that have changed and undeclares any that have been deleted. If *NAME* is NIL, *all* declarations are edited.

COM_1 ... COM_N are (optional) edit commands.

When you redeclare a global record, the translations of all expressions involving that record or any of its fields are automatically deleted from CLISPARRAY, and thus will be recomputed using the new information. If you change a *local* record declaration (see Chapter 21), or change some other CLISP declaration (see Chapter 21), e.g., STANDARD to FAST, and wish the new information to affect record expressions already translated, you must make sure the corresponding translations are removed, usually either by CLISPIFYing or using the DW edit macro.

(**RECLOOK** *RECNAME* — — — —) [Function]

Returns the entire declaration for the record named *RECNAME*; NIL if there is no record declaration with name *RECNAME*. Note that the record package maintains internal state about current record declarations, so performing destructive operations (e.g. NCONC) on the value of RECLOOK may leave the record package in an inconsistent state. To change a record declaration, use EDITREC.

(**FIELDLOOK** *FIELDNAME*) [Function]

Returns the list of declarations in which *FIELDNAME* is the name of a field.

(**RECORDFIELDNAMES** *RECORDNAME* —) [Function]

Returns the list of fields declared in record *RECORDNAME*. *RECORDNAME* may either be a name or an entire declaration.

(**RECORDACCESS** *FIELD DATUM DEC TYPE NEWVALUE*) [Function]

TYPE is one of FETCH, REPLACE, FFETCH, FREPLACE, /REPLACE or their lowercase equivalents. *TYPE*=NIL means FETCH. If *TYPE* corresponds to a fetch operation, i.e. is FETCH, or FFETCH, RECORDACCESS performs (*TYPE FIELD* of *DATUM*). If *TYPE* corresponds to a replace, RECORDACCESS performs (*TYPE FIELD* of *DATUM* with *NEWVALUE*). *DEC* is an optional declaration; if given, *FIELD* is interpreted as a field name of that declaration.

Note that RECORDACCESS is relatively inefficient, although it is better than constructing the equivalent form and performing an EVAL.

(**RECORDACCESSFORM** *FIELD DATUM TYPE NEWVALUE*) [Function]

Returns the form that would be compiled as a result of a record access. *TYPE* is one of FETCH, REPLACE, FFETCH, FREPLACE, /REPLACE or their lowercase equivalents. *TYPE*=NIL means FETCH.

Changetran

Often, you'll want to assign a new value to some datum that is a function of its current value:

Incrementing a counter: (SETQ X (IPLUS X 1))

Pushing an item on the front of a list: (SETQ X (CONS Y X))

Popping an item off a list: (PROG1 (CAR X) (SETQ X (CDR X)))

Those are simple when you're working with a variable; it gets complicated when you're working with structured data. For example, if you want to modify (CAR X), the above examples would be:

```
(CAR (RPLACA X (IPLUS (CAR X) 1)))
(CAR (RPLACA X (CONS Y (CAR X)))
(PROG1 (CAAR X) (RPLACA X (CDAR X)))
```

and if you're changing an element in an array, (ELT A N), the examples would be:

```
(SETA A N (IPLUS (ELT A N) 1)))
(SETA A N (CONS Y (ELT A N))))
(PROG1 (CAR (ELT A N)) (SETA A N (CDR (ELT A N))))
```

Changetran is designed to provide a simpler way to express these common (but user-extensible) structure modifications. Changetran defines a set of CLISP words that encode the kind of modification to take place—pushing on a list, adding to a number, etc. More important, you only indicate the item to be modified once. Thus, the “change word” ADD is used to increase the value of a datum by the sum of a set of numbers. Its arguments are the datum, and a set of numbers to be added to it. The datum must be a variable or an accessing expression (envolving FETCH, CAR, LAST, ELT, etc) that can be translated to the appropriate setting expression.

For example, (ADD X 1) is equivalent to:

```
(SETQ X (PLUS X 1))
```

and (ADD (CADDR X) (FOO)) is equivalent to:

```
(CAR (RPLACA (CDDR X) (PLUS (FOO) (CADDR X)))
```

If the datum is a complicated form involving function calls, such as (ELT (FOO X) (FIE Y))), Changetran goes to some lengths to make sure that those subsidiary functions are evaluated only once, even though they are used in both the setting and accessing parts of the translation. You can rely on the fact that the forms will be evaluated only as often as they appear in your expression.

For ADD and all other changewords, the lowercase version (add, etc.) may also be specified. Like other CLISP words, change words are translated using all CLISP declarations in effect (see Chapter 21).

The following is a list of those change words recognized by Changetran. Except for POP, the value of all built-in changeword forms is defined to be the new value of the datum.

(**ADD** *DATUM* $ITEM_1$ $ITEM_2$ ...) [Change Word]

Adds the specified items to the current value of the datum, stores the result back in the datum location. The translation will use IPLUS, PLUS, or FPLUS according to the CLISP declarations in effect (see Chapter 21).

(**PUSH** *DATUM* $ITEM_1$ $ITEM_2$ ...) [Change Word]

CONSes the items onto the front of the current value of the datum, and stores the result back in the datum location. For example, (PUSH X A B) would translate as (SETQ X (CONS A (CONS B X))).

(**PUSHNEW** *DATUM ITEM*) [Change Word]

Like PUSH (with only one item) except that the item is not added if it is already FMEMB of the datum's value.

Note that, whereas (CAR (PUSH X 'FOO)) will always be FOO, (CAR (PUSHNEW X 'FOO)) might be something else if FOO already existed in the middle of the list.

(**PUSHLIST** *DATUM* $ITEM_1$ $ITEM_2$ ...) [Change Word]

Similar to PUSH, except that the items are APPENDed in front of the current value of the datum. For example, (PUSHLIST X A B) translates as (SETQ X (APPEND A B X)).

(**POP** *DATUM*) [Change Word]

Returns CAR of the current value of the datum after storing its CDR into the datum. The current value is computed only once even though it is referenced twice. Note that this is the only built-in changeword for which the value of the form is not the new value of the datum.

(**SWAP** $DATUM_1$ $DATUM_2$) [Change Word]

Sets $DATUM_1$ to $DATUM_2$ and vice versa.

(**CHANGE** *DATUM FORM*) [Change Word]

This is the most flexible of all change words: You give an arbitrary form describing what the new value should be. But it still highlights the fact that structure modification is happening, and still lets the datum appear only once. CHANGE sets *DATUM* to the value of *FORM**, where *FORM** is constructed from *FORM* by substituting the datum expression for every occurrence of the symbol DATUM. For example,

```
(CHANGE (CAR X) (ITIMES DATUM 5))
```

translates as

```
(CAR (RPLACA X (ITIMES (CAR X) 5))).
```

CHANGE is useful for expressing modifications that are not built-in and are not common enough to justify defining a user-changeword.

You can define new change words. To define a change word, say sub, that subtracts items from the current value of the datum, you must put the property CLISPWORD, value (CHANGETRAN . sub) on both the upper- and lower-case versions of sub:

```
(PUTPROP 'SUB 'CLISPWORD '(CHANGETRAN . sub))
(PUTPROP 'sub 'CLISPWORD '(CHANGETRAN . sub))
```

Then, you must put (on the *lower*-case version of sub only) the property CHANGEWORD, with value *FN*. *FN* is a function that will be applied to a single argument, the whole sub form, and must return a form that Changetran can translate into an appropriate expression. This form should be a list structure with the symbol DATUM used whenever you want an accessing expression for the current value of the datum to appear. The form (DATUM← *FORM*) (note that DATUM← is a single symbol) should occur once in the expression; this specifies that an appropriate storing expression into the datum should occur at that point. For example, sub could be defined as:

```
(PUTPROP 'sub 'CHANGEWORD
         '(LAMBDA (FORM)
            (LIST 'DATUM←
                  (LIST 'IDIFFERENCE
                        'DATUM
                        (CONS 'IPLUS (CDDR FORM))))))
```

If the expression (sub (CAR X) A B) were encountered, the arguments to SUB would first be dwimified, and then the CHANGEWORD function would be passed the list (sub (CAR X) A B), and return (DATUM← (IDIFFERENCE DATUM (IPLUS A B))), which Changetran would convert to (CAR (RPLACA X (IDIFFERENCE (CAR X) (IPLUS A B)))).

Note: The sub changeword as defined above will always use IDIFFERENCE and IPLUS; add uses the correct addition operation depending on the current CLISP declarations (see Chapter 21).

Built-In and User Data Types

Medley is a system for manipulating various kinds of data; it comes with a large set of built-in data types, which you can use to represent a variety of abstract objects; you can also define additional “user data types” that you can manipulate exactly like built-in data types.

Each data type in Medley has an associated “type name,” a symbol. Some of the type names of built-in data types are: LITATOM, LISTP, STRINGP, ARRAYP, STACKP, SMALLP, FIXP, and FLOATP. For user data types, the type name is specified when the data type is created.

(**DATATYPES** —) [Function]

Returns a list of all type names currently defined.

(**USERDATATYPES**) [Function]

Returns list of names of currently declared user data types.

(**TYPENAME** *DATUM*) [Function]

Returns the type name for the data type of *DATUM*.

(**TYPENAMEP** *DATUM TYPE*) [Function]

Returns T if *DATUM* is an object with type name equal to *TYPE*, otherwise NIL.

In addition to built-in data-types like symbols, lists, arrays, etc., Medley provides a way to define completely *new* classes of objects, with a fixed number of fields determined by the definition of the data type. To define a new class of objects, you must supply a name for the new data type and specifications for each of its fields. Each field may contain either a pointer (i.e., any arbitrary Interlisp datum), an integer, a floating point number, or an *N*-bit integer.

Note: The most convenient way to define new user data types is via DATATYPE record declarations (see Chapter 8) which call the following functions.

(**DECLAREDATATYPE** *TYPENAME FIELDSPECS* — —) [Function]

Defines a new user data type, with the name *TYPENAME*. *FIELDSPECS* is a list of "field specifications." Each field specification may be one of the following:

POINTER Field may contain any Interlisp datum.

FIXP Field contains an integer.

FLOATP Field contains a floating point number.

(BITS *N*) Field contains a non-negative integer less than 2^N.

BYTE Equivalent to (BITS 8).

WORD Equivalent to (BITS 16).

SIGNEDWORD Field contains a 16 bit signed integer.

DECLAREDATATYPE returns a list of "field descriptors," one for each element of *FIELDSPECS*. A field descriptor contains information about where within the datum the field is actually stored.

If *FIELDSPECS* is NIL, *TYPENAME* is "undeclared." If *TYPENAME* is already declared as a data type, it is undeclared, and then re-declared with the new *FIELDSPECS*. An instance of a data type that has been undeclared has a type name of **DEALLOC**.

(**FETCHFIELD** *DESCRIPTOR DATUM*) [Function]

Returns the contents of the field described by *DESCRIPTOR* from *DATUM*. *DESCRIPTOR* must be a "field descriptor" as returned by DECLAREDATATYPE or GETDESCRIPTORS. If *DATUM* is not an instance of the datatype of which *DESCRIPTOR* is a descriptor, causes error Datum of incorrect type.

(**REPLACEFIELD** *DESCRIPTOR DATUM NEWVALUE*) [Function]

Store *NEWVALUE* into the field of *DATUM* described by *DESCRIPTOR*. *DESCRIPTOR* must be a field descriptor as returned by DECLAREDATATYPE. If *DATUM* is not an instance of the datatype of which *DESCRIPTOR* is a descriptor, causes error `Datum of incorrect type`. Value is *NEWVALUE*.

(**NCREATE** *TYPE OLDOBJ*) [Function]

Creates and returns a new instance of datatype *TYPE*.

If *OLDOBJ* is also a datum of datatype *TYPE*, the fields of the new object are initialized to the values of the corresponding fields in *OLDOBJ*.

NCREATE will not work for built-in datatypes, such as ARRAYP, STRINGP, etc. If *TYPE* is not the type name of a previously declared *user* data type, generates an error, `Illegal data type`.

(**GETFIELDSPECS** *TYPENAME*) [Function]

Returns a list which is EQUAL to the *FIELDSPECS* argument given to DECLAREDATATYPE for *TYPENAME*; if *TYPENAME* is not a currently declared data-type, returns NIL.

(**GETDESCRIPTORS** *TYPENAME*) [Function]

Returns a list of field descriptors, EQUAL to the *value* of DECLAREDATATYPE for *TYPENAME*. If *TYPENAME* is not an atom, (TYPENAME *TYPENAME*) is used.

You can define how a user data type prints, using DEFPRINT (see Chapter 25), how they are to be evaluated by the interpreter via DEFEVAL (see Chapter 10), and how they are to be compiled by the compiler via COMPILETYPELST (see Chapter 18).

[This page intentionally left blank]

9. LISTS AND ITERATIVE STATEMENTS

Medley gives you a large number of predicates, conditional functions, and control functions. Also, there is a complex “iterative statement” facility which allows you to easily create complex loops and iterative constructs.

Data Type Predicates

Medley provides separate functions for testing whether objects are of certain commonly-used types:

(**LITATOM** *X*) [Function]

Returns `T` if *X* is a symbol; `NIL` otherwise. Note that a number is not a symbol.

(**SMALLP** *X*) [Function]

Returns *X* if *X* is a small integer; `NIL` otherwise. (The range of small integers is -65536 to +65535.

(**FIXP** *X*) [Function]

Returns *X* if *X* is a small or large integer; `NIL` otherwise.

(**FLOATP** *X*) [Function]

Returns *X* if *X* is a floating point number; `NIL` otherwise.

(**NUMBERP** *X*) [Function]

Returns *X* if *X* is a number of any type, `NIL` otherwise.

(**ATOM** *X*) [Function]

Returns `T` if *X* is an atom (i.e. a symbol or a number); `NIL` otherwise.

`(ATOM X)` is `NIL` if *X* is an array, string, etc. In Common Lisp, `CL:ATOM` is defined equivalent to the Interlisp function `NLISTP`.

(**LISTP** *X*) [Function]

Returns *X* if *X* is a list cell (something created by `CONS`); `NIL` otherwise.

(**NLISTP** *X*) [Function]

`(NOT (LISTP X))`. Returns `T` if *X* is not a list cell, `NIL` otherwise.

(**STRINGP** *X*) [Function]

Returns *X* if *X* is a string, `NIL` otherwise.

(**ARRAYP** *X*) [Function]

Returns *X* if *X* is an array, `NIL` otherwise.

(**HARRAYP** *X*) [Function]

Returns *X* if it is a hash array object; otherwise `NIL`.

HARRAYP returns NIL if *X* is a list whose CAR is an HARRAYP, even though this is accepted by the hash array functions.

Note: The empty list, () or NIL, is considered to be a symbol, rather than a list. Therefore, (LITATOM NIL) = (ATOM NIL) = T and (LISTP NIL) = NIL. Take care when using these functions if the object may be the empty list NIL.

Equality Predicates

Sometimes, there is more than one type of equality. For instance, given two lists, you can ask whether they are exactly the same object, or whether they are two distinct lists that contain the same elements. Confusion between these two types of equality is often the source of program errors.

(**EQ** *X Y*) [Function]

Returns T if *X* and *Y* are identical pointers; NIL otherwise. EQ should not be used to compare two numbers, unless they are small integers; use EQP instead.

(**NEQ** *X Y*) [Function]

The same as (NOT (EQ *X* *Y*))

(**NULL** *X*) [Function]
(**NOT** *X*) [Function]

The same as (EQ *X* NIL)

(**EQP** *X Y*) [Function]

Returns T if *X* and *Y* are EQ, or if *X* and *Y* are numbers and are equal in value; NIL otherwise. For more discussion of EQP and other number functions, see Chapter 7.

EQP also can be used to compare stack pointers (Section 11) and compiled code (Chapter 10).

(**EQUAL** *X Y*) [Function]

EQUAL returns T if *X* and *Y* are one of the following:

1. EQ
2. EQP, i.e., numbers with equal value
3. STREQUAL, i.e., strings containing the same sequence of characters
4. Lists and CAR of *X* is EQUAL to CAR of *Y*, and CDR of *X* is EQUAL to CDR of *Y*

EQUAL returns NIL otherwise. Note that EQUAL can be significantly slower than EQ.

A loose description of EQUAL might be to say that *X* and *Y* are EQUAL if they print out the same way.

(**EQUALALL** *X Y*) [Function]

Like EQUAL, except it descends into the contents of arrays, hash arrays, user data types, etc. Two non-EQ arrays may be EQUALALL if their respective componants are EQUALALL.

Note: In general, EQUALALL descends all the way into all datatypes, both those you've defined and those built into the system. If you have a data structure with fonts and pointers to windows, EQUALALL will descend those also. If the data structures are circular, as windows are, EQUALALL can cause stack overflow.

Logical Predicates

(**AND** X_1 X_2 ... X_N) [NLambda NoSpread Function]

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument evaluates to NIL, AND immediately returns NIL, without evaluating the remaining arguments. If all of the arguments evaluate to non-NIL, the value of the last argument is returned. (AND) => T.

(**OR** X_1 X_2 ... X_N) [NLambda NoSpread Function]

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument is non-NIL, the value of that argument is returned by OR (without evaluating the remaining arguments). If all of the arguments evaluate to NIL, NIL is returned. (OR) => NIL.

AND and OR can be used as simple logical connectives, but note that they may not evaluate all of their arguments. This makes a difference if some of the arguments cause side-effects. This also means you can use AND and OR as simple conditional statements. For example: (AND (LISTP *X*) (CDR *X*)) returns the value of (CDR *X*) if *X* is a list cell; otherwise it returns NIL without evaluating (CDR *X*). In general, you should avoid this use of AND and OR in favor of more explicit conditional statements in order to make programs more readable.

COND Conditional Function

(**COND** $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$) [NLambda NoSpread Function]

COND takes an indefinite number of arguments, called clauses. Each $CLAUSE_i$ is a list of the form (P_i C_{i1} ... C_{iN}), where P_i is the predicate, and C_{i1} ... C_{iN} are the consequents. The operation of COND can be paraphrased as:

IF P_1 THEN C_{11} ... C_{1N} ELSEIF P_2 THEN C_{21} ... C_{2N} ELSEIF P_3 ...

The clauses are considered in sequence as follows: The predicate P_1 of the clause $CLAUSE_i$ is evaluated. If the value of P_1 is "true" (non-NIL), the consequents C_{i1} ... C_{iN} are evaluated in order, and the value of the COND is the value of the last expression in the clause. If P_1 is "false" (EQ to NIL), then the remainder of $CLAUSE_i$ is ignored, and the next clause, $CLAUSE_{i+1}$, is considered. If no P_i is true for *any* clause, the value of the COND is NIL.

If a clause has no consequents, and has the form (P_i), then if P_i evaluates to non-NIL, it is returned as the value of the COND. It is only evaluated once.

Example:

```
←(DEFINEQ (DOUBLE (X)
   (COND ((NUMBERP X) (PLUS X X))
```

```
    ((STRINGP X) (CONCAT X X))
    ((ATOM X) (PACK* X X))
    (T (PRINT "unknown") X)
    ((HORRIBLE-ERROR))]
 (DOUBLE)
←(DOUBLE 5)
   10
←(DOUBLE "FOO")
   "FOOFOO"
←(DOUBLE 'BAR)
   BARBAR
←(DOUBLE '(A B C))
   "unknown"
   (A B C)
```

A few points about this example: Notice that 5 is both a number and an atom, but it is "caught" by the NUMBERP clause before the ATOM clause. Also notice the predicate T, which is always true. This is the normal way to indicate a COND clause which will always be executed (if none of the preceeding clauses are true). (HORRIBLE-ERROR) will never be executed.

The IF Statement

The IF statement lets you write conditional expressions that are easier to read than using COND directly. CLISP translates expressions using IF, THEN, ELSEIF, or ELSE (or their lowercase versions) into equivalent CONDs. In general, statements of the form:

(if *AAA* then *BBB* elseif *CCC* then *DDD* else *EEE*)

are translated to:

```
(COND (AAA BBB)
      (CCC DDD)
      (T EEE))
```

The segment between IF or ELSEIF and the next THEN corresponds to the predicate of a COND clause, and the segment between THEN and the next ELSE or ELSEIF as the consequent(s). ELSE is the same as ELSEIF T THEN. These words are spelling corrected using the spelling list CLISPIFWORDSPLST. You may also use lower-case versions (if, then, elseif, else).

If there is nothing following a THEN, or THEN is omitted entirely, the resulting COND clause has a predicate but no consequent. For example, (if X then elseif ...) and (if X elseif ...) both translate to (COND (X) ...)—if X is not NIL, it is returned as the value of the COND.

Each predicate must be a single expression, but multiple expressions are allowed as the consequents after THEN or ELSE. Multiple consequent expressions are implicitely wrapped in a PROGN, and the value of the last one is returned as the value of the consequent. For example:

```
(if X then (PRINT "FOO") (PRINT "BAR") elseif Y then (PRINT "BAZ"))
```

Selection Functions

(**SELECTQ** *X* $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$ *DEFAULT*) [NLambda NoSpread Function]

Selects a form or sequence of forms based on the value of *X*. Each clause $CLAUSE_i$ is a list of the form (S_i C_{i1} **...** C_{iN}) where S_i is the selection key. Think of SELECTQ as:

IF *X* = S_1 THEN C_{11} ... C_{1N} ELSEIF *X* = S_2
THEN ... ELSE *DEFAULT*

If S_i is a symbol, the value of *X* is tested to see if it is EQ to S_i (which is *not* evaluated). If so, the expressions C_{i1} ... C_{iN} are evaluated in sequence, and the value of the SELECTQ is the value of the last expression.

If S_i is a list, the value of *X* is compared with each element (not evaluated) of S_i, and if *X* is EQ to any one of them, then C_{i1} ... C_{iN} are evaluated as above.

If $CLAUSE_i$ is not selected in one of the two ways described, $CLAUSE_{i+1}$ is tested, etc., until all the clauses have been tested. If none is selected, *DEFAULT* is evaluated, and its value is returned as the value of the SELECTQ. *DEFAULT* must be present.

An example of the form of a SELECTQ is:

```
[SELECTQ MONTH
    (FEBRUARY (if (LEAPYEARP) then 29 else 28))
        ((SEPTEMBER APRIL JUNE NOVEMBER) 30) 31]
```

If the value of MONTH is the symbol FEBRUARY, the SELECTQ returns 28 or 29 (depending on (LEAPYEARP)); otherwise if MONTH is APRIL, JUNE, SEPTEMBER, or NOVEMBER, the SELECTQ returns 30; otherwise it returns 31.

SELECTQ compiles open, and is therefore very fast; however, it will not work if the value of *X* is a list, a large integer, or floating point number, since SELECTQ uses EQ for all comparisons.

SELCHARQ (Chapter 2) is a version of SELECTQ that recognizes CHARCODE symbols.

(**SELECTC** *X* $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$ *DEFAULT*) [NLambda NoSpread Function]

"SELECTQ-on-Constant." Like SELECTQ, but the selection keys are evaluated, and the result used as a SELECTQ-style selection key.

SELECTC is compiled as a SELECTQ, with the selection keys evaluated at compile-time. Therefore, the selection keys act like compile-time constants (see Chapter 18).

For example:

```
[SELECTC NUM
  ((for X from 1 to 9 collect (TIMES X X)) "SQUARE") "HIP"]
```

compiles as:

```
(SELECTQ NUM
  ((1 4 9 16 25 36 49 64 81) "SQUARE") "HIP")
```

PROG and Associated Control Functions

(**PROG1** X_1 X_2 ... X_N) [NLambda NoSpread Function]

Evaluates its arguments in order, and returns the value of its first argument X_1. For example, `(PROG1 X (SETQ X Y))` sets `X` to `Y`, and returns `X`'s original value.

(**PROG2** X_1 X_2 ... X_N) [NoSpread Function]

Like `PROG1`. Evaluates its arguments in order, and returns the value of its second argument X_2.

(**PROGN** X_1 X_2 ... X_N) [NLambda NoSpread Function]

`PROGN` evaluates each of its arguments in order, and returns the value of its last argument. `PROGN` is used to specify more than one computation where the syntax allows only one, e.g., `(SELECTQ ... (PROGN ...))` allows evaluation of several expressions as the default condition for a `SELECTQ`.

(**PROG** *VARLST* E_1 E_2 ... E_N) [NLambda NoSpread Function]

Lets you bind some variables while you execute a series of expressions. *VARLST* is a list of local variables (must be `NIL` if no variables are used). Each symbol in *VARLST* is treated as the name of a local variable and bound to `NIL`. *VARLST* can also contain lists of the form (*NAME FORM*). In this case, *NAME* is the name of the variable and is bound to the value of *FORM*. The evaluation takes place before any of the bindings are performed, e.g., `(PROG ((X Y) (Y X)) ...)` will bind local variable `X` to the value of `Y` (evaluated *outside* the `PROG`) and local variable `Y` to the value of `X` (outside the `PROG`). An attempt to use anything other than a symbol as a `PROG` variable will cause an error, `Arg not symbol`. An attempt to use `NIL` or `T` as a `PROG` variable will cause an error, `Attempt to bind NIL or T`.

The rest of the `PROG` is a sequence of forms and symbols (labels). The forms are evaluated sequentially; the labels serve only as markers. The two special functions, `GO` and `RETURN`, alter this flow of control as described below. The value of the `PROG` is usually specified by the function `RETURN`. If no `RETURN` is executed before the `PROG` "falls off the end," the value of the `PROG` is `NIL`.

(**GO** *L*) [NLambda NoSpread Function]

`GO` is used to cause a transfer in a `PROG`. (`GO` *L*) will cause the `PROG` to evaluate forms starting at the label *L* (`GO` does not evaluate its argument). A `GO` can be used at any level in a `PROG`. If the label is not found, `GO` will search higher progs *within the same function*, e.g., `(PROG ... A ... (PROG ... (GO A)))`. If the label is not found in the function in which the `PROG` appears, an error is generated, `Undefined or illegal GO`.

(**RETURN** *X*) [Function]

A `RETURN` is the normal exit for a `PROG`. Its argument is evaluated and is immediately returned the value of the `PROG` in which it appears.

Note: If a GO or RETURN is executed in an interpreted function which is not a PROG, the GO or RETURN will be executed in the last interpreted PROG entered if any, otherwise cause an error.

GO or RETURN inside of a compiled function that is not a PROG is not allowed, and will cause an error at compile time.

As a corollary, GO or RETURN in a functional argument, e.g., to SORT, will not work compiled. Also, since NLSETQ's and ERSETQ's compile as *separate* functions, a GO or RETURN *cannot* be used inside of a compiled NLSETQ or ERSETQ if the corresponding PROG is outside, i.e., above, the NLSETQ or ERSETQ.

(**LET** *VARLST* E_1 E_2 . . . E_N) [Macro]

LET is essentially a PROG that can't contain GO's or RETURN's, and whose last form is the returned value.

(**LET*** *VARLST* E_1 E_2 . . . E_N) [Macro]
(**PROG*** *VARLST* E_1 E_2 . . . E_N) [Macro]

LET* and PROG* differ from LET and PROG only in that the binding of the bound variables is done "sequentially." Thus

```
(LET* ((A (LIST 5))
       (B (LIST A A)))
      (EQ A (CADR B)))
```

would evaluate to T; whereas the same form with LET might find A an unbound variable when evaluating (LIST A A).

The Iterative Statement

The various forms of the iterative statement (i.s.) let you write complex loops easily. Rather than writing PROG, MAPC, MAPCAR, etc., let Medley do it for you.

An iterative statement is a form consisting of a number of special words (known as i.s. operators or i.s.oprs), followed by operands. Many i.s.oprs (FOR, DO, WHILE, etc.) act like loops in other programming languages; others (COLLECT, JOIN, IN, etc.) do things useful in Lisp. You can also use lower-case versions of i.s.oprs (do, collect, etc.).

```
← (for X from 1 to 5 do (PRINT 'FOO))
FOO
FOO
FOO
FOO
FOO
NIL
←(for X from 2 to 10 by 2 collect (TIMES X X))
(4 16 36 64 100)
←(for X in '(A B 1 C 6.5 NIL (45)) count (NUMBERP X))
2
```

Iterative statements are implemented using CLISP, which translates them into the appropriate PROGs, MAPCARs, etc. They're are translated using all CLISP declarations in effect (standard / fast / undoable /

etc.); see Chapter 21. Misspelled i.s.oprs are recognized and corrected using the spelling list CLISPFORWORDSPLST. Operators can appear in any order; CLISP scans the entire statement before it begins to translate.

If you define a function with the same name as an i.s.opr (WHILE, TO, etc.), that i.s.opr will no longer cause looping when it appears as CAR of a form, although it will continue to be treated as an i.s.opr if it appears in the interior of an iterative statement. To alert you, a warning message is printed, e.g., (While defined, therefore disabled in CLISP).

I.S. Types

Every iterative statement must have exactly one of the following operators in it (its "is.stype"), to specify what happens on each iteration. Its operand is called the "body" of the iterative statement.

DO *FORMS* [I.S. Operator]

Evaluate *FORMS* at each iteration. DO with no other operator specifies an infinite loop. If some explicit or implicit terminating condition is specified, the value of the loop is NIL. Translates to MAPC or MAP whenever possible.

COLLECT *FORM* [I.S. Operator]

The value of *FORM* at each iteration is collected in a list, which is returned as the value of the loop when it terminates. Translates to MAPCAR, MAPLIST or SUBSET whenever possible.

When COLLECT translates to a PROG (if UNTIL, WHILE, etc. appear in the loop), the translation employs an open TCONC using two pointers similar to that used by the compiler for compiling MAPCAR. To disable this translation, perform (CLDISABLE 'FCOLLECT).

JOIN *FORM* [I.S. Operator]

FORM returns a list; the lists from each iteration are concatenated using NCONC, forming one long list. Translates to MAPCONC or MAPCON whenever possible. /NCONC, /MAPCONC, and /MAPCON are used when the CLISP declaration UNDOABLE is in effect.

SUM *FORM* [I.S. Operator]

The values of *FORM* from each iteration are added together and returned as the value of the loop, e.g., (for I from 1 to 5 sum (TIMES I I)) returns 1+4+9+16+25 = 55. IPLUS, FPLUS, or PLUS will be used in the translation depending on the CLISP declarations in effect.

COUNT *FORM* [I.S. Operator]

Counts the number of times that *FORM* is true, and returns that count as the loop's value.

ALWAYS *FORM* [I.S. Operator]

Returns T if the value of *FORM* is non-NIL for all iterations. **Note**: Returns NIL as soon as the value of *FORM* is NIL).

NEVER *FORM* [I.S. Operator]

Like `ALWAYS`, but returns `T` if the value of *FORM* is *never* true. **Note**: Returns `NIL` as soon as the value of *FORM* is non-`NIL`.

Often, you'll want to set a variable each time through the loop; that's called the "iteration variable", or i.v. for short. The following i.s.types explicitly refer to the i.v. This is explained below under `FOR`.

THEREIS *FORM* [I.S. Operator]

Returns the first value of the i.v. for which *FORM* is non-`NIL`, e.g., `(for X in Y thereis (NUMBERP X))` returns the first number in `Y`.

Note: Returns the value of the i.v. as soon as the value of *FORM* is non-`NIL`.

LARGEST *FORM* [I.S. Operator]
SMALLEST *FORM* [I.S. Operator]

Returns the value of the i.v. that provides the largest/smallest value of *FORM*. `$$EXTREME` is always bound to the current greatest/smallest value, `$$VAL` to the value of the i.v. from which it came.

Iteration Variable I.s.oprs

You'll want to bind variables to use during the loop. Rather than putting the loop inside a `PROG` or `LET`, you can specify bindings like so:

BIND *VAR* [I.S. Operator]
BIND *VARS* [I.S. Operator]

Used to specify dummy variables, which are bound locally within the i.s.

Note: You can initialize a variable *VAR* by saying *VAR*←*FORM*:

```
(bind HEIGHT ← 0 WEIGHT ← 0 for SOLDIER in ...)
```

To specify iteration variables, use these operators:

FOR *VAR* [I.S. Operator]

Specifies the iteration variable (i.v.) that is used in conjunction with `IN`, `ON`, `FROM`, `TO`, and `BY`. The variable is rebound within the loop, so the value of the variable outside the loop is not affected. Example:

```
←(SETQ X 55)
55
←(for X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
←X
55
```

FOR OLD *VAR* [I.S. Operator]

Like `FOR`, but *VAR* is *not* rebound, so its value outside the loop *is* changed. Example:

```
←(SETQ X 55)
55
```

```
←(for old X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
←X
6
```

FOR *VARS* [I.S. Operator]

VARS a list of variables, e.g., `(for (X Y Z) in ...)`. The first variable is the i.v., the rest are dummy variables. See `BIND` above.

IN *FORM* [I.S. Operator]

FORM must evaluate to a list. The i.v. is set to successive elements of the list, one per iteration. For example, `(for X in Y do ...)` corresponds to `(MAPC Y (FUNCTION (LAMBDA (X) ...)))`. If no i.v. has been specified, a dummy is supplied, e.g., `(in Y collect CADR)` is equivalent to `(MAPCAR Y (FUNCTION CADR))`.

ON *FORM* [I.S. Operator]

Same as `IN`, but the i.v. is reset to the corresponding *tail* at each iteration. Thus `IN` corresponds to `MAPC`, `MAPCAR`, and `MAPCONC`, while `ON` corresponds to `MAP`, `MAPLIST`, and `MAPCON`.

```
←(for X on '(A B C) do (PRINT X))
(A B C)
(B C)
(C)
NIL
```

Note: For both `IN` and `ON`, *FORM* is evaluated before the main part of the i.s. is entered, i.e. *outside* of the scope of any of the bound variables of the i.s. For example, `(for X bind (Y←'(1 2 3)) in Y ...)` will map down the list which is the value of `Y` evaluated *outside* of the i.s., *not* `(1 2 3)`.

IN OLD *VAR* [I.S. Operator]

Specifies that the i.s. is to iterate down *VAR*, with *VAR* itself being reset to the corresponding tail at each iteration, e.g., after `(for X in old L do ... until ...)` finishes, `L` will be some tail of its original value.

IN OLD (*VAR*←*FORM*) [I.S. Operator]

Same as `IN OLD` *VAR*, except *VAR* is first set to value of *FORM*.

ON OLD *VAR* [I.S. Operator]

Same as `IN OLD` *VAR* except the i.v. is reset to the current value of *VAR* at each iteration, instead of to `(CAR` *VAR*`)`.

ON OLD (*VAR*←*FORM*) [I.S. Operator]

Same as `ON OLD` *VAR*, except *VAR* is first set to value of *FORM*.

INSIDE *FORM* [I.S. Operator]

Like `IN`, but treats first non-list, non-`NIL` tail as the last element of the iteration, e.g., `INSIDE '(A B C D . E)` iterates five times with the i.v. set to `E` on the last iteration. `INSIDE 'A` is equivalent to `INSIDE '(A)`, which will iterate once.

FROM *FORM* [I.S. Operator]

Specifies the initial value for a numerical i.v. The i.v. is automatically incremented by 1 after each iteration (unless `BY` is specified). If no i.v. has been specified, a dummy i.v. is supplied and initialized, e.g., `(from 2 to 5 collect SQRT)` returns `(1.414 1.732 2.0 2.236)`.

TO *FORM* [I.S. Operator]

Specifies the final value for a numerical i.v. If `FROM` is not specified, the i.v. is initialized to 1. If no i.v. has been specified, a dummy i.v. is supplied and initialized. If `BY` is not specified, the i.v. is automatically incremented by 1 after each iteration. When the i.v. is definitely being *incremented*, i.e., either `BY` is not specified, or its operand is a positive number, the i.s. terminates when the i.v. exceeds the value of *FORM*. Similarly, when the i.v. is definitely being decremented the i.s. terminates when the i.v. becomes *less* than the value of *FORM* (see description of `BY`).

FORM is evaluated only once, when the i.s. is first entered, and its value bound to a temporary variable against which the i.v. is checked each interation. If the user wishes to specify an i.s. in which the value of the boundary condition is recomputed each iteration, he should use `WHILE` or `UNTIL` instead of `TO`.

When both the operands to `TO` and `FROM` are numbers, and `TO`'s operand is less than `FROM`'s operand, the i.v. is decremented by 1 after each iteration. In this case, the i.s. terminates when the i.v. becomes *less* than the value of *FORM*. For example, `(from 10 to 1 do PRINT)` prints the numbers from `10` down to `1`.

BY *FORM* (without `IN` or `ON`) [I.S. Operator]

If you aren't using `IN` or `ON`, `BY` specifies how the i.v. itself is reset at each iteration. If you're using `FROM` or `TO`, the i.v. is known to be numerical, so the new i.v. is computed by adding the value of *FORM* (which is reevaluated each iteration) to the current value of the i.v., e.g., `(for N from 1 to 10 by 2 collect N)` makes a list of the first five odd numbers.

If *FORM* is a positive number (*FORM* itself, not its value, which in general CLISP would have no way of knowing in advance), the loop stops when the value of the i.v. *exceeds* the value of `TO`'s operand. If *FORM* is a negative number, the loop stops when the value of the i.v. becomes *less* than `TO`'s operand, e.g., `(for I from N to M by -2 until (LESSP I M) ...)`. Otherwise, the terminating condition for each iteration depends on the value of *FORM* for that iteration: if *FORM*`<0`, the test is whether the i.v. is less than `TO`'s operand, if *FORM*`>0` the test is whether the i.v. exceeds `TO`'s operand; if *FORM* `= 0`, the loop terminates unconditionally.

If you didn't use FROM or TO and *FORM* is not a number, the i.v. is simply reset to the value of *FORM* after each iteration, e.g., `(for I from N by (FOO) ...)` sets I to the value of `(FOO)` on each loop after the first.

BY *FORM* (with IN or ON) [I.S. Operator]

If you did use IN or ON, *FORM*'s value determines the *tail* for the next iteration, which in turn determines the value for the i.v. as described earlier, i.e., the new i.v. is CAR of the tail for IN, the tail itself for ON. In conjunction with IN, you can refer to the current tail within *FORM* by using the i.v. or the operand for IN/ON, e.g., `(for Z in L by (CDDR Z) ...)` or `(for Z in L by (CDDR L) ...)`. At translation time, the name of the internal variable which holds the value of the current tail is substituted for the i.v. throughout *FORM*. For example, `(for X in Y by (CDR (MEMB 'FOO (CDR X))) collect X)` specifies that after each iteration, CDR of the current tail is to be searched for the atom FOO, and (CDR of) this latter tail to be used for the next iteration.

AS *VAR* [I.S. Operator]

Lets you have more than one i.v. for a single loop, e.g., `(for X in Y as U in V do ...)` moves through the lisps Y and V in parallel (see MAP2C). The loop ends when any of the terminating conditions is met, e.g., `(for X in Y as I from 1 to 10 collect X)` makes a list of the first ten elements of Y, or however many elements there are on Y if less than 10.

The operand to AS, *VAR*, specifies the new i.v. For the remainder of the i.s., or until another AS is encountered, all operators refer to the new i.v. For example, (for I from 1 to N_1 as J from 1 to N_2 by 2 as K from N_3 to 1 by -1 ...) terminates when I exceeds N_1, or J exceeds N_2, or K becomes less than 1. After each iteration, I is incremented by 1, J by 2, and K by -1.

OUTOF *FORM* [I.S. Operator]

For use with generators. On each iteration, the i.v. is set to successive values returned by the generator. The loop ends when the generator runs out.

Condition I.S. Oprs

What if you want to do things only on certain times through the loop? You could make the loop body a big COND, but it's much more readable to use one of these:

WHEN *FORM* [I.S. Operator]

Only run the loop body when FORM's value is non-NIL. For example, `(for X in Y collect X when (NUMBERP X))` collects only the elements of Y that are numbers.

UNLESS *FORM* [I.S. Operator]

Opposite of WHEN: WHEN Z is the same as `UNLESS (NOT Z)`.

WHILE *FORM* [I.S. Operator]

WHILE *FORM* evaluates *FORM* *before* each iteration, and if the value is NIL, exits.

UNTIL *FORM* [I.S. Operator]

Opposite of WHILE: Evaluates *FORM before* each iteration, and if the value is *not* NIL, exits.

REPEATWHILE *FORM* [I.S. Operator]

Same as WHILE except the test is performed *after* the loop body, but before the i.v. is reset for the next iteration.

REPEATUNTIL *FORM* [I.S. Operator]

Same as UNTIL, except the test is performed *after* the loop body.

Other I.S. Operators

FIRST *FORM* [I.S. Operator]

FORM is evaluated once before the first iteration, e.g., (for X Y Z in L first (FOO Y Z) ...), and FOO could be used to initialize Y and Z.

FINALLY *FORM* [I.S. Operator]

FORM is evaluated after the loop terminates. For example, (for X in L bind Y_0 do (if (ATOM X) then (SETQ Y (PLUS Y 1))) finally (RETURN Y)) will return the number of atoms in L.

EACHTIME *FORM* [I.S. Operator]

FORM is evaluated at the beginning of each iteration before, and regardless of, any testing. For example, consider,

```
(for I from 1 to N
   do (... (FOO I) ...)
   unless (... (FOO I) ...)
   until (... (FOO I) ...))
```

You might want to set a temporary variable to the value of (FOO I) in order to avoid computing it three times each iteration. However, without knowing the translation, you can't know whether to put the assignment in the operand to DO, UNLESS, or UNTIL. You can avoid this problem by simply writing EACHTIME (SETQ J (FOO I)).

DECLARE: *DECL* [I.S. Operator]

Inserts the form (DECLARE *DECL*) immediately following the PROG variable list in the translation, or, in the case that the translation is a mapping function rather than a PROG, immediately following the argument list of the lambda expression in the translation. This can be used to declare variables bound in the iterative statement to be compiled as local or special variables. For example (for X in Y declare: (LOCALVARS X) ...). Several DECLARE:s can apppear in the same i.s.; the declarations are inserted in the order they appear.

DECLARE *DECL* [I.S. Operator]

Same as DECLARE:.

Since DECLARE is also the name of a function, DECLARE cannot be used as an i.s. operator when it appears as CAR of a form, i.e. as the first i.s. operator in an iterative statement. However, declare (lowercase version) *can* be the first i.s. operator.

ORIGINAL *I.S.OPR OPERAND* [I.S. Operator]

I.S.OPR will be translated using its original, built-in interpretation, independent of any user defined i.s. operators.

There are also a number of i.s.oprs that make it easier to create iterative statements that use the clock, looping for a given period of time. See timers, Chapter 12.

Miscellaneous Hints For Using I.S.Oprs

Lowercase versions of all i.s. operators are equivalent to the uppercase, e.g., (for X in Y ...) is equivalent to (FOR X IN Y ...).

Each i.s. operator is of lower precedence than all Interlisp forms, so parentheses around the operands can be omitted, and will be supplied where necessary, e.g., BIND (X Y Z) can be written BIND X Y Z, OLD (X_*FORM*) as OLD X_*FORM*, etc.

RETURN or GO may be used in any operand. (In this case, the translation of the iterative statement will always be in the form of a PROG, never a mapping function.) RETURN means return from the loop (with the indicated value), *not* from the function in which the loop appears. GO refers to a label elsewhere in the function in which the loop. appears, except for the labels $$LP, $$ITERATE, and $$OUT which are reserved, as described below.

In the case of FIRST, FINALLY, EACHTIME, DECLARE: or one of the i.s.types, e.g., DO, COLLECT, SUM, etc., the operand can consist of more than one form, e.g., COLLECT (PRINT (CAR X)) (CDR X), in which case a PROGN is supplied.

Each operand can be the name of a function, in which case it is applied to the (last) i.v., e.g., (for X in Y do PRINT when NUMBERP) is the same as (for X in Y do (PRINT X) when (NUMBERP X)). Note that the i.v. need not be explicitly specified, e.g., (in Y do PRINT when NUMBERP) will work.

For i.s.types, e.g., DO, COLLECT, JOIN, the function is always applied to the first i.v. in the i.s., whether explicity named or not. For example, (in Y as I from 1 to 10 do PRINT) prints elements on Y, not integers between 1 and 10.

Note that this feature does not make much sense for FOR, OLD, BIND, IN, or ON, since they “operate” before the loop starts, when the i.v. may not even be bound.

In the case of BY in conjunction with IN, the function is applied to the current *tail* e.g., (for X in Y by CDDR ...) is the same as (for X in Y by (CDDR X) ...).

While the exact translation of a loop depends on which operators are present, a PROG will always be used whenever the loop specifies dummy variables—if BIND appears, or there is more than one variable specified by a FOR, or a GO, RETURN, or a reference to the variable $$VAL appears in any of the operands. When PROG is used, the form of the translation is:

```
(PROG VARIABLES
   {initialize}
```

```
$$LP {eachtime}
    {test}
    {body}
$$ITERATE
    {aftertest}
    {update}
    (GO $$LP)
$$OUT {finalize}
    (RETURN $$VAL))
```

where {test} corresponds to that part of the loop that tests for termination and also for those iterations for which {body} is not going to be executed, (as indicated by a WHEN or UNLESS); {body} corresponds to the operand of the i.s.type, e.g., DO, COLLECT, etc.; {aftertest} corresponds to those tests for termination specified by REPEATWHILE or REPEATUNTIL; and {update} corresponds to that part that resets the tail, increments the counter, etc. in preparation for the next iteration. {initialize}, {finalize}, and {eachtime} correspond to the operands of FIRST, FINALLY, and EACHTIME, if any.

Since {body} always appears at the top level of the PROG, you can insert labels in {body}, and GO to them from within {body} or from other i.s. operands, e.g., (for X in Y first (GO A) do (FOO) A (FIE)). However, since {body} is dwimified as a list of forms, the label(s) should be added to the dummy variables for the iterative statement in order to prevent their being dwimified and possibly “corrected”, e.g., (for X in Y bind A first (GO A) do (FOO) A (FIE)). You can also GO to $$LP, $$ITERATE, or $$OUT, or explicitly set $$VAL.

Errors in Iterative Statements

An error will be generated and an appropriate diagnostic printed if any of the following conditions hold:

1. Operator with null operand, i.e., two adjacent operators, as in (for X in Y until do ...)
2. Operand consisting of more than one form (except as operand to FIRST, FINALLY, or one of the i.s.types), e.g., (for X in Y (PRINT X) collect ...).
3. IN, ON, FROM, TO, or BY appear twice in same i.s.
4. Both IN and ON used on same i.v.
5. FROM or TO used with IN or ON on same i.v.
6. More than one i.s.type, e.g., a DO and a SUM.

In 3, 4, or 5, an error is not generated if an intervening AS occurs.

If an error occurs, the i.s. is left unchanged.

If no DO, COLLECT, JOIN or any of the other i.s.types are specified, CLISP will first attempt to find an operand consisting of more than one form, e.g., (for X in Y (PRINT X) when ATOM X ...), and in this case will insert a DO after the first form. (In this case, condition 2 is not considered to be met, and an error is not generated.) If CLISP cannot find such an operand, and no WHILE or UNTIL appears in the i.s., a warning message is printed: NO DO, COLLECT, OR JOIN: followed by the i.s.

Similarly, if no terminating condition is detected, i.e., no IN, ON, WHILE, UNTIL, TO, or a RETURN or GO, a warning message is printed: Possible non-terminating iterative statement: followed by the iterative statement. However, since the user may be planning to terminate the i.s. via an error, Control-E, or a RETFROM from a lower function, the i.s. is still translated.

Note: The error message is not printed if the value of CLISPI.S.GAG is T (initially NIL).

Defining New Iterative Statement Operators

The following function is available for defining new or redefining existing iterative statement operators:

(**I.S.OPR** *NAME FORM OTHERS EVALFLG*) [Function]

NAME is the name of the new i.s.opr. If *FORM* is a list, *NAME* will be a new *i.s.type*, and *FORM* its body.

OTHERS is an (optional) list of additional i.s. operators and operands which will be added to the i.s. at the place where *NAME* appears. If *FORM* is NIL, *NAME* is a new i.s.opr defined entirely by *OTHERS*.

In both *FORM* and *OTHERS*, the atom $$VAL can be used to reference the value to be returned by the i.s., I.V. to reference the current i.v., and BODY to reference *NAME*'s operand. In other words, the current i.v. will be substituted for all instances of I.V. and *NAME*'s operand will be substituted for all instances of BODY throughout *FORM* and *OTHERS*.

If *EVALFLG* is T, *FORM* and *OTHERS* are evaluated at translation time, and their values used as described above. A dummy variable for use in translation that does not clash with a dummy variable already used by some other i.s. operators can be obtained by calling (GETDUMMYVAR). (GETDUMMYVAR T) will return a dummy variable and also insure that it is bound as a PROG variable in the translation.

If *NAME* was previously an i.s.opr and is being redefined, the message (*NAME* REDEFINED) will be printed (unless DFNFLG=T), and all expressions using the i.s.opr *NAME* that have been translated will have their translations discarded.

The following are some examples of how I.S.OPR could be called to define some existing i.s.oprs, and create some new ones:

```
COLLECT  (I.S.OPR 'COLLECT
            '(SETQ $$VAL (NCONC1 $$VAL BODY)))

    SUM  (I.S.OPR 'SUM
            '(SETQ $$VAL_(PLUS $$VAL BODY)
                '(FIRST (SETQ $$VAL0))

  NEVER  (I.S.OPR 'NEVER
            '(if BODY then
                (SETQ $$VAL NIL) (GO $$OUT))
```

Note: (if BODY then (RETURN NIL)) would exit from the i.s. immediately and therefore not execute the operations specified via a FINALLY (if any).

THEREIS
```
(I.S.OPR 'THEREIS
  '(if BODY then
     (SETQ $$VAL I.V.) (GO $$OUT)))
```

RCOLLECT To define RCOLLECT, a version of COLLECT which uses CONS instead of NCONC1 and then reverses the list of values:

```
(I.S.OPR 'RCOLLECT
  '(FINALLY (RETURN
     (DREVERSE $$VAL)))]
```

TCOLLECT To define TCOLLECT, a version of COLLECT which uses TCONC:

```
(I.S.OPR 'TCOLLECT
  '(TCONC $$VAL BODY)
     '(FIRST (SETQ $$VAL (CONS))
        FINALLY (RETURN
           (CAR $$VAL)))]
```

PRODUCT
```
(I.S.OPR 'PRODUCT
  '(SETQ $$VAL $$VAL*BODY)
     '(FIRST ($$VAL 1))]
```

UPTO To define UPTO, a version of TO whose operand is evaluated only once:

```
(I.S.OPR 'UPTO
  NIL
  '(BIND $$FOO←BODY TO $$FOO)]
```

TO To redefine TO so that instead of recomputing *FORM* each iteration, a variable is bound to the value of *FORM*, and then that variable is used:

```
(I.S.OPR 'TO
  NIL
  '(BIND $$END FIRST
     (SETQ $$END BODY)
        ORIGINALTO $$END)]
```

Note the use of ORIGINAL to redefine TO in terms of its original definition. ORIGINAL is intended for use in redefining built-in operators, since their definitions are not accessible, and hence not directly modifiable. Thus if the operator had been defined by the user via I.S.OPR, ORIGINAL would not obtain its original definition. In this case, one presumably would simply modify the i.s.opr definition.

I.S.OPR can also be used to define synonyms for already defined i.s. operators by calling I.S.OPR with *FORM* an atom, e.g., (I.S.OPR 'WHERE 'WHEN) makes WHERE be the same as WHEN. Similarly, following (I.S.OPR 'ISTHERE 'THEREIS), one can write (ISTHERE ATOM IN Y), and following (I.S.OPR 'FIND 'FOR) and (I.S.OPR 'SUCHTHAT 'THEREIS), one can write (find X in Y suchthat X member Z) . In the current system, WHERE is synonymous with WHEN, SUCHTHAT and ISTHERE with THEREIS, FIND with FOR, and THRU with TO.

If *FORM* is the atom MODIFIER, then *NAME* is defined as an i.s.opr which can immediately follow another i.s. operator (i.e., an error will not be generated, as described previously). *NAME* will not terminate the scope of the previous operator, and will be stripped off when DWIMIFY is called on its operand. OLD is an example of a MODIFIER type of operator. The MODIFIER feature allows the user to define i.s. operators similar to OLD, for use in conjunction with some other user defined i.s.opr which will produce the appropriate translation.

The file package command I.S.OPRS (Chapter 17) will dump the definition of i.s.oprs. (I.S.OPRS PRODUCT UPTO) as a file package command will print suitable expressions so that these iterative statement operators will be (re)defined when the file is loaded.

[This page intentionally left blank]

FUNCTION DEFINITION, MANIPULATION AND EVALUATION

Medley is designed to help you define and debug functions. Developing an applications program with Medley involves defining a number of functions in terms of the system primitives and other user-defined functions. Once defined, your functions may be used exactly like Interlisp primitive functions, so the programming process can be viewed as extending the Interlisp language to include the required functionality.

A function's definition specifies if the function has a fixed or variable number of arguments, whether these arguments are evaluated or not, the function argument names, and a series of forms which define the behavior of the function. For example:

```
(LAMBDA (X Y) (PRINT X) (PRINT Y))
```

This function has two evaluated arguments, X and Y, and it will execute (PRINT X) and (PRINT Y) when evaluated. Other types of function definitions are described below.

A function is defined by putting an expr definition in the function definition cell of a symbol. There are a number of functions for accessing and setting function definition cells, but one usually defines a function with DEFINEQ (see the Defining Functions section below). For example:

```
← (DEFINEQ (FOO (LAMBDA (X Y) (PRINT X) (PRINT Y))))(FOO)
```

The expression above will define the function FOO to have the expr definition (LAMBDA (X Y) (PRINT X) (PRINT Y)). After being defined, this function may be evaluated just like any system function:

```
← (FOO 3 (IPLUS 3 4))
    3
    7
    7
```

Not all function definition cells contain expr definitions. The compiler (see the first page of Chapter 18) translates expr definitions into compiled code objects, which execute much faster. Interlisp provides a number of "function type functions" which determine how a given function is defined, the number and names of function arguments, etc. See the Function Type Functions section below.

Usually, functions are evaluated automatically when they appear within another function or when typed into Interlisp. However, sometimes it is useful to envoke the Interlisp interpreter explicitly to apply a given "functional argument" to some data. There are a number of functions which will apply a given function repeatedly. For example, MAPCAR will apply a function (or an expr definition) to all of the elements of a list, and return the values returned by the function:

```
← (MAPCAR '(1 2 3 4 5) '(LAMBDA (X) (ITIMES X X))
    (1 4 9 16 25)
```

When using functional arguments, there are a number of problems which can arise, related to accessing free variables from within a function argument. Many times these problems can be solved using the function FUNCTION to create a FUNARG object.

The macro facility provides another way of specifying the behavior of a function (see the Macros section below). Macros are very useful when developing code which should run very quickly, which should be compiled differently than when it is interpreted, or which should run differently in different implementations of Interlisp.

Function Types

Interlisp functions are defined using list expressions called "expr definitions." An expr definition is a list of the form (*LAMBDA-WORD ARG-LIST* $FORM_1$ ... $FORM_N$). *LAMBDA-WORD* determines whether the arguments to this function will be evaluated or not. *ARG-LIST* determines the number and names of arguments. $FORM_1$ ... $FORM_N$ are a series of forms to be evaluated after the arguments are bound to the local variables in *ARG-LIST*.

If *LAMBDA-WORD* is the symbol LAMBDA, then the arguments to the function are evaluated. If *LAMBDA-WORD* is the symbol NLAMBDA, then the arguments to the function are not evaluated. Functions which evaluate or don't evaluate their arguments are therefore known as "lambda" or "nlambda" functions, respectively.

If *ARG-LIST* is NIL or a list of symbols, this indicates a function with a fixed number of arguments. Each symbol is the name of an argument for the function defined by this expression. The process of binding these symbols to the individual arguments is called "spreading" the arguments, and the function is called a "spread" function. If the argument list is any symbol other than NIL, this indicates a function with a variable number of arguments, known as a "nospread" function.

If *ARG-LIST* is anything other than a symbol or a list of symbols, such as (LAMBDA "FOO" ...), attempting to use this expr definition will generate an Arg not symbol error. In addition, if NIL or T is used as an argument name, the error Attempt to bind NIL or T is generated.

These two parameters (lambda / nlambda and spread / nospread) may be specified independently, so there are four nain function types, known as lambda-spread, nlanbda-spread, lanbda-nospread, and nlambda-nospread functions. Each one has a different form and is used for a different purpose. These four function types are described more fully below.

For lambda-spread, lanbda-nospread, or nlambda-spread functions, there is an upper limit to the number of arguments that a function can have, based on the number of arguments that can be stored on the stack on any one function call. Currently, the limit is 80 arguments. If a function is called with more than that many arguments, the error Too many arguments occurs. However, nlambda-nospread functions can be called with an arbitrary number of arguments, since the arguments are not individually saved on the stack.

Lambda-Spread Functions

Lambda-spread functions take a fixed number of evaluated arguments. This is the most common function type. A lambda-spread expr definition has the form:

(LAMBDA (ARG_1 ... ARG_M) $FORM_1$ ... $FORM_N$)

The argument list (ARG_1 ... ARG_M) is a list of symbols that gives the number and names of the formal arguments to the function. If the argument list is ( ) or NIL, this indicates that the function takes no arguments. When a lambda-spread function is applied to some arguments, the arguments are evaluated, and bound to the local variables ARG_1 ... ARG_M. Then, $FORM_1$ ... $FORM_N$ are evaluated in order, and the value of the function is the value of $FORM_N$.

```
← (DEFINEQ (FOO (LAMBDA (X Y) (LIST X Y))))
    (FOO)
← (FOO 99 (PLUS 3 4))
    (99 7)
```

In the above example, the function FOO defined by (LAMBDA (X Y) (LIST X Y)) is applied to the arguments 99 and (PLUS 3 4). These arguments are evaluated (giving 99 and 7), the local variable X is bound to 99 and Y to 7, (LIST X Y) is evaluated, returning (99 7), and this is returned as the value of the function.

A standard feature of the Interlisp system is that no error occurs if a spread function is called with too many or too few arguments. If a function is called with too many argumnents, the extra arguments are evaluated but ignored. If a function is called with too few arguments, the unsupplied ones will be delivered as NIL. In fact, a spread function cannot distinguish between being given NIL as an argument, and not being given that argument, e.g., (FOO) and (FOO NIL) are exactly the same for spread functions. If it is necessary to distinguish between these two cases, use an nlambda function and explicitly evaluate the arguments with the EVAL function.

Nlambda-Spread Functions

Nlambda-spread functions take a fixed number of unevaluated arguments. An nlambda-spread expr definition has the form:

(NLAMBDA *(ARG*$_1$ *... ARG*$_M$) *FORM*$_1$ *... FORM*$_N$)

Nlambda-spread functions are evaluated similarly to lanbda-spread functions, except that the arguments are not evaluated before being bound to the variables *ARG*$_1$ *... ARG*$_M$.

```
← (DEFINEQ (FOO (NLAMBDA (X Y) (LIST X Y))))
  (FOO)
← (FOO 99 (PLUS 3 4))
  (99 (PLUS 3 4))
```

In the above example, the function FOO defined by (NLAMBDA (X Y) (LIST X Y)) is applied to the arguments 99 and (PLUS 3 4). These arguments are unevaluated to X and Y. (LIST X Y) is evaluated, returning (99 (PLUS 3 4)), and this is returned as the value of the function.

Functions can be defined so that all of their arguments are evaluated (lambda functions) or none are evaluated (nlambda functions). If it is desirable to write a function which only evaluates some of its arguments (e.g., SETQ), the functions should be defined as an nlambda, with some arguments explicitly evaluated using the function EVAL. If this is done, the user should put the symbol EVAL on the property list of the function under the property INFO. This informs various system packages, such as DWIM, CLISP, and Masterscope, that this function in fact does evaluate its arguments, even though it is an nlambda.

Warning: A frequent problem that occurs when evaluating arguments to nlambda functions with EVAL is that the form being evaluated may reference variables that are not accessible within the nlambda function. This is usually not a problem when interpreting code, but when the code is compiled, the values of “local” variables may not be accessible on the stack (see Chapter 18). The system nlambda functions that evaluate their arguments (such as SETQ) are expanded in-line by the compiler, so this is not a problem. Using the macro facility is recommended in cases where it is necessary to evaluate some arguments to an nlambda function.

Lambda-Nospread Functions

Lambda-nospread functions take a variable number of evaluated arguments. A lambda-nospread expr definition has the form:

(LAMBDA *VAR FORM*$_1$ *... FORM*$_N$)

VAR may be any symbol, except NIL and T. When a lambda-nospread function is applied to some arguments, each of these arguments is evaluated and the values stored on the stack. *VAR* is then bound to the number of arguments which have been evaluated. For example, if FOO is defined by (LAMBDA X ...), when (FOO A B C) is evaluated, A, B, and C are evaluated and X is bound to 3. *VAR* should never be reset

The following functions are used for accessing the arguments of lambda-nospread functions.

(**ARG** *VAR M*) [NLambda Function]

Returns the *M*th argument for the lambda-nospread function whose argument list is *VAR*. *VAR* is the name of the atomic argument list to a lambda-nospread function, and is not evaluated. *M* is the number of the desired argument, and is evaluated. The value of ARG is undefined for *M* less than or equal to 0 or greater than the value of *VAR*.

(**SETARG** *VAR M X*) [NLambda Function]

Sets the *M*th argument for the lambda-nospread function whose argument list is *VAR* to *X*. *VAR* is not evaluated; *M* and *X* are evaluated. *M* should be between 1 and the value of *VAR*.

In the example below, the function FOO is defined to collect and return a list of all of the evaluated arguments it is given (the value of the for statement).

```
← (DEFINEQ (FOO
  (LAMBDA X (for ARGNUM from 1 to X collect (ARG X ARGNUM)]
   (FOO)
← (FOO 99 (PLUS 3 4))
   (99 7)
← (FOO 99 (PLUS 3 4)(TIMES 3 4)))
   (99 7 12)
```

NLambda-Nospread Functions

Nlambda-nospread functions take a variable number of unevaluated arguments. An nlambda-nospread expr definition has the form:

(NLAMBDA *VAR* $FORM_1$ ... $FORM_N$)

VAR may be any symbol, except NIL and T. Though similar in form to lambda-nospread expr definitions, an nlambda-nospread is evaluated quite differently. When an nlambda-nospread function is applied to some arguments, *VAR* is simply bound to a list of the unevaluated arguments. The user may pick apart this list, and evaluate different arguments.

In the example below, FOO is defined to return the reverse of the list of arguments it is given (unevaluated):

```
← (DEFINEQ (FOO (NLAMBDA X (REVERSE X))))
   (FOO)
← (FOO 99 (PLUS 3 4))
   ((PLUS 3 4) 99)
← (FOO 99 (PLUS 3 4)(TIMES 3 4))
   (TIMES 3 4)(PLUS 3 4) 99)
```

The warning about evaluating arguments to nlambda functions also applies to nlambda-nospread function.

Compiled Functions

Functions defined by expr definitions can be compiled by the Interlisp compiler (see Chapter 18). The compiler produces compiled code objects (of data type CCODEP) which execute more quickly than the corresponding expr definition code. Functions defined by compiled code objects may have the same four types as expr definitions (lambda / nlambda, spread / nospread). Functions created by the compiler are referred to as compiled functions.

Function Type Functions

There are a variety of functions used for examining the type, argument list, etc. of functions. These functions may be given either a symbol (in which case they obtain the function definition from the definition cell), or a function definition itself.

(**FNTYP** *FN*) [Function]

Returns NIL if *FN* is not a function definition or the name of a defined function. Otherwise, FNTYP returns one of the following symbols, depending on the type of function definition.

EXPR Lambda-spread expr definition

CEXPR Lambda-spread compiled definition

FEXPR Nlambda-spread expr definition

CFEXPR Nlambda-spread compiled definition

EXPR* Lambda-nospread expr definition

CEXPR* Lambda-nospread compiled definition

FEXPR* Nlambda-nospread expr definition

CFEXPR* Nlambda-nospread compiled definition

FUNARG FNTYP returns the symbol FUNARG if *FN* is a FUNARG expression.

EXP, FEXPR, EXPR*, and FEXPR* indicate that *FN* is defined by an expr definition. CEXPR, CFEXPR, CEXPR*, and CFEXPR* indicate that *FN* is defined by a compiled definition, as indicated by the prefix C. The suffix * indicates that *FN* has an indefinite number of arguments, i.e., is a nospread function. The prefix F indicates unevaluated arguments. Thus, for example, a CFEXPR* is a compiled nospread nlambda function.

(**EXPRP** *FN*) [Function]

Returns T if (FNTYP *FN*) is EXPR, FEXPR, EXPR*, or FEXPR*; NIL otherwise. However, (EXPRP *FN*) is also true if *FN* is (has) a list definition, even if it does not begin with LAMBDA or NLAMBDA. In other words, EXPRP is not quite as selective as FNTYP.

(**CCODEP** *FN*) [Function]

Returns T if (FNTYP *FN*) is either CEXPR, CFEXPR, CEXPR*, or CFEXPR*; NIL otherwise.

(**ARGTYPE** *FN*) [Function]

FN is the name of a function or its definition. ARGTYPE returns 0, 1, 2, or 3, or NIL if *FN* is not a function. ARGTYPE corresponds to the rows of FNTYPs. The interpretation of this value is as follows:

0 Lambda-spread function (EXPR, CEXPR)

1 Nlambda-spread function (FEXPR, CFEXPR)

2 Lambda-nospread function (EXPR*, CEXPR*)

3 Nlambda-nospread function (FEXPR*, CFEXPR*)

(**NARGS** *FN*) [Function]

Returns the number of arguments of *FN*, or NIL if *FN* is not a function. If *FN* is a nospread function, the value of NARGS is 1.

(**ARGLIST** *FN*) [Function]

Returns the "argument list" for *FN*. Note that the "argument list" is a symbol for nospread functions. Since NIL is a possible value for ARGLIST, the error Args not available is generated if *FN* is not a function.

If *FN* is a compiled function, the argument list is constructed, i.e., each call to ARGLIST requires making a new list. For functions defined by expr definitions, lists beginning with LAMBDA or NLAMBDA, the argument list is simply CADR of GETD. If *FN* has an expr definition, and CAR of the definition is not LAMBDA or NLAMBDA, ARGLIST will check to see if CAR of the definition is a member of LAMBDASPLST (see Chapter 20). If it is, ARGLIST presumes this is a function object the user is defining via DWIMUSERFORMS, and simply returns CADR of the definition as its argument list. Otherwise ARGLIST generates an error as described above.

(**SMARTARGLIST** *FN EXPLAINFLG TAIL*) [Function]

A "smart" version of ARGLIST that tries various strategies to get the arglist of *FN*.

First SMARTARGLIST checks the property list of *FN* under the property ARGNAMES. For spread functions, the argument list itself is stored. For nospread functions, the form is (NIL ARGLIST_1 . ARGLIST_2), where ARGLIST_1 is the value SMARTARGLIST should return when *EXPLAINFLG* = T, and ARGLIST_2 the value when *EXPLAINFLG* = NIL. For example, (GETPROP 'DEFINEQ 'ARGNAMES) = (NIL (X1 Xl ... XN) . X). This allows the user to specify special argument lists.

Second, if *FN* is not defined as a function, SMARTARGLIST attempts spelling correction on *FN* by calling FNCHECK (see Chapter 20), passing *TAIL* to be used for the call to FIXSPELL. If unsuccessful, the FN Not a function error will be generated.

Third, if *FN* is known to the file package (see Chapter 17) but not loaded in, SMARTARGLIST will obtain the arglist information from the file.

Otherwise, SMARTARGLIST simply returns (ARGLIST *FN*).

SMARTARGLIST is used by BREAK (see Chapter 15) and ADVISE with *EXPLAINFLG* = NIL for constructing equivalent expr definitions, and by the TTYIN in-line command ?= (see Chapter 26), with *EXPLAINFLG* = T.

Defining Functions

Function definitions are stored in a "function definition cell" associated with each symbol. This cell is directly accessible via the two functions PUTD and GETD (see below), but it is usually easier to define functions with DEFINEQ:

(**DEFINEQ** *X1 X2 . . . XN*) [NLambda NoSpread Function]

DEFINEQ is the function normally used for defining functions. It takes an indefinite number of arguments which are not evaluated. Each X_i must be a list defining one function, of the form (NAME DEFINITION). For example:

```
(DEFINEQ (DOUBLE (LAMBDA (X) (IPLUS X X))))
```

The above expression will define the function DOUBLE with the expr definition (LAMBDA (X) (IPLUS X X)). X_i may also have the form (NAME ARGS . DEF-BODY), in which case an appropriate lambda expr definition will be constructed. Therefore, the above expression is exactly the same as:

```
(DEFINEQ (DOUBLE (X) (IPLUS X X)))
```

Note that this alternate form can only be used for lambda functions. The first form must be used to define an nlambda function.

DEFINEQ returns a list of the names of the functions defined.

(**DEFINE** *X* —) [Function]

Lambda-spread version of DEFINEQ. Each element of the list *X* is itself a list either of the form (NAME DEFINITION) or (NAME ARGS . DEF-BODY). DEFINE will generate an error, Incorrect defining form on encountering an atom where a defining list is expected.

DEFINE and DEFINEQ operate correctly if the function is already defined and BROKEN, ADVISED, or BROKEN-IN.

For expressions involving type-in only, if the time stamp facility is enabled (see the Time Stamps section of Chapter 16), both DEFINE and DEFINEQ stamp the definition with your initials and date.

UNSAFE.TO.MODIFY.FNS [Variable]

Value is a list of functions that should not be redefined, because doing so may cause unusual bugs (or crash the system!). If you try to modify a function on this list (using DEFINEQ, TRACE, etc), the system prints Warning: XXX may be unsafe to modify -- continue? If you type Yes, the function is modified, otherwise an error occurs. This provides a measure of safety for novices who may accidently redefine important system functions. You can add your own functions onto this list.

By convention, all functions starting with the character backslash ("\") are system internal functions, which you should never redefine or modify. Backslash functions are not on UNSAFE.TO.MODIFY.FNS, so trying to redefine them will not cause a warning.

DFNFLG [Variable]

DFNFLG is a global variable that affects the operation of DEFINEQ and DEFINE. If DFNFLG=NIL, an attempt to *redefine* a function FN will cause DEFINE to print the message (FN REDEFINED) and to save the old definition of FN using SAVEDEF (see the Functions for Manipulating Typed Definitions section of Chapter 17) before redefining it (except if the old and new definitions are EQUAL, in which case the effect is simply a no-op). If DFNFLG=T, the function is simply redefined. If DFNFLG=PROP or ALLPROP, the new definition is stored on the property list under the property EXPR. ALLPROP also affects the operation of RPAQQ and RPAQ (see the Functions Used Within Source Files section of Chapter 17). DFNFLG is initially NIL.

DFNFLG is reset by LOAD (see the Loading Files section of Chapter 17) to enable various ways of handling the defining of functions and setting of variables when loading a file. For most applications, the user will not reset DFNFLG directly.

Note: The compiler does *not* respect the value of DFNFLG when it redefines functions to their compiled definitions (see the first page of Chapter 18). Therefore, if you set DFNFLG to PROP to completely avoid inadvertantly redefining something in your running system, you *must* use compile mode F, not ST.

Note that the functions SAVEDEF and UNSAVEDEF (see the Functions for Manipulating Typed Definitions section of Chapter 17) can be useful for "saving" and restoring function definitions from property lists.

(**GETD** *FN*) [Function]

Returns the function definition of *FN*. Returns NIL if *FN* is not a symbol, or has no definition.

GETD of a compiled function constructs a pointer to the definition, with the result that two successive calls do not necessarily produce EQ results. EQP or EQUAL must be used to compare compiled definitions.

(**PUTD** *FN DEF* —) [Function]

Puts *DEF* into *FN*'s function cell, and returns *DEF*. Generates an error, Arg not symbol, if FN is not a symbol. Generates an error, Illegal arg, if *DEF* is a string, number, or a symbol other than NIL.

(**MOVD** *FROM TO COPYFLG* —) [Function]

Moves the definition of *FROM* to *TO*, i.e., redefines *TO*. If *COPYFLG* = T, a COPY of the definition of *FROM* is used. *COPYFLG* =T is only meaningful for expr definitions, although MOVD works for compiled functions as well. MOVD returns *TO*.

COPYDEF (see the Functions for Manipulating Typed Definitions section of Chapter 17) is a higher-level function that not only moves expr definitions, but works also for variables, records, etc.

(**MOVD?** *FROM TO COPYFLG* —) [Function]

If *TO* is not defined, same as (MOVD *FROM TO COPYFLG*). Otherwise, does nothing and returns NIL.

Function Evaluation

Usually, function application is done automatically by the Interlisp interpreter. If a form is typed into Interlisp whose CAR is a function, this function is applied to the arguments in the CDR of the form. These arguments are evaluated or not, and bound to the funcion parameters, as determined by the type of the function, and the body of the function is evaluated. This sequence is repeated as each form in the body of the function is evaluated.

There are some situations where it is necessary to explicitly call the evaluator, and Interlisp supplies a number of functions that will do this. These functions take "functional arguments," which may either be symbols with function definitions, or expr definition forms such as (LAMBDA (X)...), or FUNARG expressions.

(**APPLY** *FN ARGLIST* —) [Function]

Applies the function *FN* to the arguments in the list *ARGLIST*, and returns its value. APPLY is a lambda function, so its arguments are evaluated, but the individual elements of *ARGLIST* are not evaluated. Therefore, lambda and nlambda functions are treated the same by APPLY—lambda functions take their arguments from *ARGLIST* without evaluating them. For example:

```
← (APPLY 'APPEND '((PLUS 1 2 3)(4 5 6)))
    (PLUS 1 2 3 4 5 6)
```

Note that *FN* may explicitly evaluate one or more of its arguments itself. For example, the system function SETQ is an nlambda function that explicitly evaluates its second argument. Therefore, (APPLY 'SETQ '(FOO (ADD1 3))) will set FOO to 4, instead of setting it to the expression (ADD1 3).

APPLY can be used for manipulating expr definitions. For example:

```
← (APPLY '(LAMBDA (X Y)(ITIMES X Y)) '(3 4)))
    12
```

(**APPLY*** *FN* ARG_1 ARG_2 ... ARG_N) [NoSpread Function]

Nospread version of APPLY. Applies the function *FN* to the arguments ARG_1 ARG_2 ... ARG_N. For example:

```
← (APPLY 'APPEND '(PLUS 1 2 3)(4 5 6))
    (PLUS 1 2 3 4 5 6)
```

(**EVAL** *X*—) [Function]

EVAL evaluates the expression *X* and returns this value, i.e., EVAL provides a way of calling the Interlisp interpreter. Note that EVAL is itself a lambda function, so its argument is first evaluated, e.g.:

```
← (SETQ FOO 'ADD1 3)))
    (ADD1 3)
←(EVAL FOO)
    4
←(EVAL 'FOO)
    (ADD1 3)
```

(**QUOTE** *X*) [Nlambda NoSpread Function]

QUOTE prevents its arguments from being evaluated. Its value is *X* itself, e.g., (QUOTE FOO) is FOO.

Interlisp functions can either evaluate or not evaluate their arguments. QUOTE can be used in those cases where it is desirable to specify arguments unevaluated.

The single-quote character (') is defined with a read macro so it returns the next expression, wrapped in a call to QUOTE (see Chapter 25). For example, 'FOO reads as (QUOTE FOO). This is the form used for examples in this manual.

Since giving QUOTE more than one argument is almost always a parenthese error, and one that would otherewise go undetected, QUOTE itself generates an error in this case, Parenthesis error.

(**KWOTE** *X*) [Function]

Value is an expression which, when evaluated, yields *X*. If *X* is NIL or a number, this is *X* itself. Otherwise (LIST (QUOTE QUOTE) X). For example:

```
(KWOTE 5) => 5
(KWOTE (CONS 'A 'B)) => (QUOTE (A.B))
```

(**NLAMBDA.ARGS** *X*) [Function]

This function interprets its argument as a list of unevaluated nlambda arguments. If any of the elements in this list are of the form (QUOTE...), the enclosing QUOTE is stripped off. Actually, NLAMBDA.ARGS stops processing the list after the first non-quoted argument. Therefore, whereas (NLAMBDA.ARGS '((QUOTE FOO) BAR)) -> (FOO BAR), (NLAMBDA.ARGS '(FOO (QUOTE BAR))) -> (FOO (QUOTE BAR)).

NLAMBDA.ARGS is alled by a number of nlambda functions in the system, to interpret their arguments. For instance, the function BREAK calls NLAMBDA.ARGS so that (BREAK 'FOO) will break the function FOO, rather than the function QUOTE.

(**EVALA** *X A*) [Function]

Simulates association list variable lookup. *X* is a form, *A* is a list of the form:

(($NAME_1$. VAL_1) ($NAME_2$. VAL_2)... ($NAME_N$. VAL_N))

The variable names and values in *A* are "spread" on the stack, and then *X* is evaluated. Therefore, any variables appearing free in *X* that also appears as CAR of an element of *A* will be given the value on the CDR of that element.

(**DEFEVAL** *TYPE FN*) [Function]

Specifies how a datum of a particular type is to be evaluated. Intended primarily for user-defined data types, but works for all data types except lists, literal atoms, and numbers. *TYPE* is a type name. *FN* is a function object, i.e., name of a function or a lambda expression. Whenever the interpreter encounters a datum of the indicated type, *FN* is applied to the datum and its value returned as the result of the evaluation. DEFEVAL returns the previous evaling function for this type. If *FN* = NIL, DEFEVAL returns the current evaling function without changing it. If *FN* = T, the evaling functions is set back to the system default (which for all data types except lists is to return the datum itself).

COMPILETYPELST (see Chapter 18) permits the user to specify how a datum of a particular type is to be compiled.

(**EVALHOOK** *FORM EVALHOOKFN*) [Function]

EVALHOOK evaluates the expression *FORM*, and returns its value. While evaluating *FORM*, the function EVAL behaves in a special way. Whenever a list other than *FORM* itself is to be evaluated, whether implicitly or via an explicit call to EVAL, *EVALHOOKFN* is invoked (it should be a function), with the form to be evaluated as its argument. *EVALHOOKFN* is then responsible for evaluating the form. Whatever is returned is assume to be the result of evaluating the form. During the execution of *EVALHOOKFN*, this special evaluation is turned off. (Note that EVALHOOK does not affect the evaluations of variables, only of lists).

Here is an example of a simple tracing routine that uses the EVALHOOK feature:

```
←(DEFINEQ (PRINTHOOK (FORM)
   (printout T "eval: "FORM T)
   (EVALHOOK FORM (FUNCTION PRINTHOOK
   (PRINTHOOK)
```

Using PRINTHOOK, one might see the following interaction:

```
←(EVALHOOK '(LIST (CONS 1 2)(CONS 3 4)) 'PRINTHOOK)
   eval: (CONS 1 2)
   eval: (CONS 3 4)
   ((1.2)(3.4))
```

Iterating and Mapping Functions

The functions below are used to evaluate a form or apply a function repeatedly. RPT, RPTQ, and FRPTQ evaluate an expression a specified number of time. MAP, MAPCAR, MAPLIST, etc., apply a given function repeatedly to different elements of a list, possibly constructing another list.

These functions allow efficient iterative computations, but they are difficult to use. For programming iterative computations, it is usually better to use the CLISP Iterative Statement facility (see Chapter 9), which provides a more general and complete facility for expressing iterative statements. Whenever possible, CLISP transltes iterative statements into expressions using the functions below, so there is no efficiency loss.

(**RPT** *N FORM*) [Function]

Evaluates the expression *FORM*, *N* times. Returns the value of the last evaluation. If *N* is less than or equal to 0, *FORM* is not evaluated, and RPT returns NIL.

Before each evaluation, the local variable RPTN is bound to the number of evaluations yet to take place. This variable can be referenced within *FORM*. For example, (RPT 10 '(PRINT RPTN)) will print the numbers 10, 9...1, and return 1.

(**RPTQ** *N* $FORM_1$ $FORM_2$ ... $FORM_N$) [NLambda NoSpread Function]

Nlambda-nospread version of RPT: *N* is evaluated, $FORM_i$ are not. Returns the value of the last evaluation of $FORM_N$.

(**FRPTQ** *N* $FORM_1$ $FORM_2$ ... $FORM_N$) [NLambda NoSpread Function]

Faster version of RPTQ. Does not bind RPTN.

(**MAP** MAP_X $MAPFN_1$ $MAPFN_2$) [Function]

If $MAPFN_2$ is NIL, MAP applies the function $MAPFN_1$ to successive tails of the list MAP_X. That is, first it computes ($MAPFN_1$ MAP_X), and then ($MAPFN_1$ (CDR MAP_X)), etc., until MAP_X becomes a non-list. If $MAPFN_2$ is provided, ($MAPFN_2$ MAP_X) is used instead of (CDR MAP_X) for the next call for $MAPFN_1$, e.g., if $MAPFN_2$ were CDDR, alternate elements of the list would be skipped. MAP returns NIL.

(**MAPC** MAP_X $MAPFN_1$ $MAPFN_2$) [Function]

Identical to MAP, except that ($MAPFN_1$ (CAR MAP_X)) is computed at each iteration instead of ($MAPFN_1$ MAP_X), i.e., MAPC works on elements, MAP on tails. MAPC returns NIL.

(**MAPLIST** *MAP*$_X$ *MAPFN*$_1$ *MAPFN*$_2$) [Function]

Successively computes the same values that MAP would compute, and returns a list consisting of those values.

(**MAPCAR** *MAP*$_X$ *MAPFN*$_1$ *MAPFN*$_2$) [Function]

Computes the same values that MAPC would compute, and returns a list consisting of those values, e.g., (MAPCAR X 'FNTYP) is a list of FNTYPs for each element on X.

(**MAPCON** *MAP*$_X$ *MAPFN*$_1$ *MAPFN*$_2$) [Function]

Computes the same values that MAP and MAPLIST but NCONCs these values to form a list which it returns.

(**MAPCONC** *MAP*$_X$ *MAPFN*$_1$ *MAPFN*$_2$) [Function]

Computes the same values that MAPC and MAPCAR, but NCONCs the values to form a list which it returns.

Note that MAPCAR creates a new list which is a mapping of the old list in that each element of the new list is the result of applying a function to the corresponding element on the original list. MAPCONC is used when there are a variable number of elements (including none) to be inserted at each iteration. Examples:

```
(MAPCONC '(A B C NIL D NIL) '(LAMBDA (Y)(if (NULL Y) then NIL
   else (LIST Y)))) = > (A B C D)
```

This MAPCONC returns a list consisting of MAP$_X$ with all NILs removed.

```
(MAPCONC '((A B) C (D E F)(G) H I) '(LAMBDA (Y)(if (LISP Y) then Y
   else NIL))) = > (A B D E F G)
```

This MAPCONC returns a linear list consisting of all the lists on *MAP*$_X$.

Since MAPCONC uses NCONC to string the corresponding lists together, in this example the original list will be altered to be ((A B C D E F G) C (D E F G)(G) H I). If this is an undesirable side effect, the functional argument to MAPCONC should return instead a top level copy of the lists, i.e., (LAMBDA (Y) (if (LISTP Y) then (APPERND Y) else NIL))).

(**MAP2C** *MAP*$_X$ *MAP*$_Y$ *MAPFN*$_1$ *MAPFN*$_2$) [Function]

Identical to MAPC except *MAPFN*$_1$ is a function of two arguments, and (*MAPFN*$_1$ (CAR *MAP*$_X$) (CAR *MAP*$_Y$)) is computed at each iteration. Terminates when either *MAP*$_X$ or *MAP*$_Y$ is a non-list.

MAPFN$_2$ is still a function of one argument, and is applied twice on each iteration; (*MAPFN*$_2$ *MAP*$_X$) gives the new *MAP*$_X$, (*MAPFN*$_2$ *MAP*$_Y$) the new *MAP*$_Y$. CDR is used if *MAPFN*$_2$ is not supplied, i.e., is NIL.

(**MAP2CAR** *MAP*$_X$ *MAP*$_Y$ *MAPFN*$_1$ *MAPFN*$_2$) [Function]

Identical to MAPCAR except *MAPFN*$_1$ is a function of two arguments, and (*MAPFN*$_1$ (CAR *MAP*$_X$) (CAR *MAP*$_Y$)) is used to assemble the new list. Terminates when either *MAP*$_X$ or *MAP*$_Y$ is a non-list.

(**SUBSET** *MAP_X $MAPFN_1$ $MAPFN_2$*) [Function]

Applies *$MAPFN_1$* to elements of *MAP_X* and returns a list of those elements for which this application is non-NIL, e.g.:

```
(SUBSET '(A B 3 C 4) 'NUMBERP) = (3 4)
```

$MAPFN_2$ plays the same role as with MAP, MAPC, et al.

(**EVERY** *$EVERY_X$ $EVERYFN_1$ $EVERYFN_2$*) [Function]

Returns T if the result of applying *$EVERYFN_1$* to each element in *$EVERY_X$* is true, otherwise NIL. For example, (EVERY '(X Y Z) 'ATOM) => T.

EVERY operates by evaluating (*$EVERYFN_1$* (CAR *$EVERY_X$*) *$EVERY_X$*). The second argument is passed to *$EVERYFN_1$* so that it can look at the next element on *$EVERY_X$* if necessary. If *$EVERYFN_1$* yields NIL, EVERY immediately returns NIL. Otherwise, EVERY computes (*$EVERYFN_2$* *$EVERY_X$*), or (CDR *$EVERY_X$*) if *$EVERYFN_2$* = NIL, and uses this as the "new" *$EVERY_X$*, and the process continues. For example (EVERY X 'ATOM 'CDDR) is true if every other element of *X* is atomic.

(**SOME** *$SOME_X$ $SOMEFN_1$ $SOMEFN_2$*) [Function]

Returns the tail of *$SOME_X$* beginning with the first element that satisfies *$SOMEFN_1$*, i.e., for which *$SOMEFN_1$* applied to that element is true. Value is NIL if no such element exists. (SOME X '(LAMBDA (Z) (EQUAL Z Y))) is equivalent to (MEMBER Y X). SOME operates analogously to EVERY. At each stage, (*$SOMEFN_1$* (CAR *$SOME_X$*) *$SOME_X$*) is computed, and if this not NIL, *$SOME_X$* is returned as the value of SOME. Otherwise, (*$SOMEFN_2$* *$SOME_X$*) is computed, or (CDR *$SOME_X$*) if *$SOMEFN_2$* = NIL, and used for the next *$SOME_X$*.

(**NOTANY** *$SOME_X$ $SOMEFN_1$ $SOMEFN_2$*) [Function]

(NOT (SOME *$SOME_X$ $SOMEFN_1$ $SOMEFN_2$*)).

(**NOTEVERY** *$EVERY_X$ $EVERYFN_1$ $EVERYFN_2$*) [Function]

(NOT (EVERY *$EVERY_X$ $EVERYFN_1$ $EVERYFN_2$*)).

(**MAPRINT** *LST FILE LEFT RIGHT SEP PFN LISPXPRINTFLG*) [Function]

A general printing function. For each element of the list *LST*, applies *PFN* to the element, and *FILE*. If *PFN* is NIL, PRIN1 is used. Between each application MAPRINT performs PRIN1 of *SEP* (or "" if *SEP* = NIL). If *LEFT* is given, it is printed (using PRIN1) initially; if *RIGHT* is given, it is printed (using PRIN1) at the end.

For example, (MAPRINT X NIL '%('%)) is equivalent to PRIN1 for lists. To print a list with commas between each element and a final "." one could use (MAPRINT X T NIL '%. '%,).

If *LISPXPRINTFLG* = T, LISPXPRIN1 (see Chapter 13) is used instead of PRIN1.

Functional Arguments

The functions that call the Interlisp-D evaluator take "functional arguments," which may be symbols with function definitions, or expr definition forms such as `(LAMBDA (X) ...)`.

The following functions are useful when one wants to supply a functional argument which will always return `NIL`, `T`, or `0`. Note that the arguments X_1 ... X_N to these functions are evaluated, though they are not used.

(**NILL** X_1 ... X_N) [NoSpread Function]

Returns `NIL`.

(**TRUE** X_1 ... X_N) [NoSpread Function]

Returns `T`.

(**ZERO** X_1 ... X_N) [NoSpread Function]

Returns `0`.

When using expr definitions as function arguments, they should be enclosed within the function `FUNCTION` rather than `QUOTE`, so that they will be compiled as separate functions.

(**FUNCTION** *FN ENV*) [NLambda Function]

If *ENV* = `NIL`, `FUNCTION` is the same as `QUOTE`, except that it is treated differently when compiled. Consider the function definition:

```
(DEFINEQ (FOO (LST)(FIE LST (FUNCTION (LAMBDA (Z)(ITIMES Z Z))]
```

`FOO` calls the function `FIE` with the value of `LST` and the expr definition `(LAMBDA (Z)(LIST (CAR Z)))`.

If `FOO` is run interpreted, it does not make any difference whether `FUNCTION` or `QUOTE` is used. However, when `FOO` is compiled, if `FUNCTION` is used the compiler will define and compile the expr definition as an auxiliary function (see Chapter 18). The compiled expr definition will run considerably faster, which can make a big difference if it is applied repeatedly.

Compiling `FUNCTION` will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (`MAPCAR`, `MAPLIST`, etc.).

If *ENV* is not `NIL`, it can be a list of variables that are (presumably) used freely by *FN*. *ENV* can also be an atom, in which case it is evaluated, and the value interpreted as described above.

Macros

Macros provide an alternative way of specifying the action of a function. Whereas function definitions are evaluated with a "function call", which involves binding variables and other housekeeping tasks, macros are evaluated by *translating* one Interlisp form into another, which is then evaluated.

A symbol may have both a function definition and a macro definition. When a form is evaluated by the interpreter, if the `CAR` has a function definition, it is used (with a function call), otherwise if it has a macro definition, then that is used. However, when a form is compiled, the CAR is checked for a macro definition first, and only if there isn't one is the function definition compiled. This allows functions that behave differently when compiled and interpreted. For example, it is possible to define a function that, when interpreted, has a function definition that is slow and has a lot of error checks, for use when debugging a system. This function could also have a macro definition that defines a fast version of the function, which is used when the debugged system is compiled.

Macro definitions are represented by lists that are stored on the property list of a symbol. Macros are often used for functions that should be compiled differently in different Interlisp implementations, and the exact property name a macro definition is stored under determines whether it should be used in a particular implementation. The global variable MACROPROPS contains a list of all possible macro property names which should be saved by the MACROS file package command. Typical macro property names are DMACRO for Interlisp-D, 10MACRO for Interlisp-10, VAXMACRO for Interlisp-VAX, JMACRO for Interlisp-Jerico, and MACRO for "implementation independent" macros. The global variable COMPILERMACROPROPS is a list of macro property names. Interlisp determines whether a symbol has a macro definition by checking these property names, in order, and using the first non-NIL property value as the macro definition. In Interlisp-D this list contains DMACRO and MACRO in that order so that DMACROs will override the implementation-independent MACRO properties. In general, use a DMACRO property for macros that are to be used only in Interlisp-D, use 10MACRO for macros that are to be used only in Interlisp-10, and use MACRO for macros that are to affect both systems.

Macro definitions can take the following forms:

(**LAMBDA ...**)
(**NLAMBDA ...**)
A function can be made to compile open by giving it a macro definition of the form (LAMBDA ...) or (NLAMBDA ...), e.g., (LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) for ABS. The effect is as if the macro definition were written in place of the function wherever it appears in a function being compiled, i.e., it compiles as a lambda or nlambda expression. This saves the time necessary to call the function at the price of more compiled code generated in-line.

(**NIL** *EXPRESSION*)
(**LIST** *EXPRESSION*)
"Substitution" macro. Each argument in the form being evaluated or compiled is substituted for the corresponding atom in LIST, and the result of the substitution is used instead of the form. For example, if the macro definition of ADD1 is ((X) (IPLUS X 1)), then, (ADD1 (CAR Y)) is compiled as (IPLUS (CAR Y) 1).

Note that ABS could be defined by the substitution macro ((X) (COND ((GREATERP X 0) X) (T (MINUS X)))). In this case, however, (ABS (FOO X)) would compile as

```
(COND ((GREATERP (FOO X) 0)
           (FOO X))
 (T (MINUS (FOO X))))
```

and (FOO X) would be evaluated two times. (Code to evaluate (FOO X) would be generated three times.)

(**OPENLAMBDA** *ARGS BODY*)
This is a cross between substitution and LAMBDA macros. When the compiler processes an OPENLAMBDA, it attempts to substitute the actual arguments for the formals wherever this preserves the frequency and

order of evaluation that would have resulted from a LAMBDA expression, and produces a LAMBDA binding only for those that require it.

Note: OPENLAMBDA assumes that it can substitute literally the actual arguments for the formal arguments in the body of the macro if the actual is side-effect free or a constant. Thus, you should be careful to use names in *ARGS* which don't occur in *BODY* (except as variable references). For example, if FOO has a macro definition of

```
(OPENLAMBDA (ENV) (FETCH (MY-RECORD-TYPE ENV) OF BAR))
```

then (FOO NIL) will expand to

```
(FETCH (MY-RECORD-TYPE NIL) OF BAR)
```

T When a macro definition is the atom T, it means that the compiler should ignore the macro, and compile the function definition; this is a simple way of turning off other macros. For example, the user may have a function that runs in both Interlisp-D and Interlisp-10, but has a macro definition that should only be used when compiling in Interlisp-10. If the MACRO property has the macro specification, a DMACRO of T will cause it to be ignored by the Interlisp-D compiler. This DMACRO would not be necessary if the macro were specified by a 10MACRO instead of a MACRO.

(**= .** *OTHER-FUNCTION*) A simple way to tell the compiler to compile one function exactly as it would compile another. For example, when compiling in Interlisp-D, FRPLACAs are treated as RPLACAs. This is achieved by having FRPLACA have a DMACRO of (= . RPLACA).

(**LITATOM** *EXPRESSION*) If a macro definition begins with a symbol other than those given above, this allows *computation* of the Interlisp expression to be evaluated or compiled in place of the form. LITATOM is bound to the CDR of the calling form, *EXPRESSION* is evaluated, and the result of this evaluation is evaluated or compiled in place of the form. For example, LIST could be compiled using the computed macro:

```
[X (LIST 'CONS (CAR X)(AND (CDR X)(CONS 'LIST (CDR X]
```

This would cause (LIST X Y Z) to compile as (CONS X (CONS Y (CONS Z NIL))). Note the recursion in the macro expansion.

If the result of the evaluation is the symbol IGNOREMACRO, the macro is ignored and the compilation of the expression proceeds as if there were no macro definition. If the symbol in question is normally treated specially by the compiler (CAR, CDR, COND, AND, etc.), and also has a macro, if the macro expansion returns IGNOREMACRO, the symbol will still be treated specially.

In Interlisp-10, if the result of the evaluation is the atom INSTRUCTIONS, no code will be generated by the compiler. It is then assumed the evaluation was done for effect and the necessary code, if any, has been added. This is a way of giving direct instructions to the compiler if you understand it.

It is often useful, when constructing complex macro expressions, to use the BQUOTE facility (see the Read Macros section of Chapter 25).

The following function is quite useful for debugging macro definitions:

(**EXPANDMACRO** *EXP QUIETFLG* – –) [Function]

Takes a form whose CAR has a macro definition and expands the form as it would be compiled. The result is prettyprinted, unless *QUIETFLG*=T, in which case the result is simply returned.

Note: EXPANDMACRO only works on Interlisp macros. Use CL:MACROEXPAND-1 to expand Interlisp macros visible to the Common Lisp interpreter and compliler.

DEFMACRO

Macros defined with the function DEFMACRO are much like "computed" macros (see the above section), in that they are defined with a form that is evaluated, and the result of the evaluation is used (evaluated or compiled) in place of the macro call. However, DEFMACRO macros support complex argument lists with optional arguments, default values, and keyword arguments as well as argument list destructuring.

(**DEFMACRO** *NAME ARGS FORM*) [NLambda NoSpread Function]

Defines *NAME* as a macro with the arguments *ARGS* and the definition form *FORM* (*NAME*, *ARGS*, and *FORM* are unevaluated). If an expression starting with *NAME* is evaluated or compiled, arguments are bound according to *ARGS*, *FORM* is evaluated, and the value of *FORM* is evaluated or compiled instead. The interpretation of *ARGS* is described below.

Note: Like the function DEFMACRO in Common Lisp, this function currently removes any function definition for *NAME*.

ARGS is a list that defines how the argument list passed to the macro *NAME* is interpreted. Specifically, *ARGS* defines a set of variables that are set to various arguments in the macro call (unevaluated), that *FORM* can reference to construct the macro form.

In the simplest case, *ARGS* is a simple list of variable names that are set to the corresponding elements of the macro call (unevaluated). For example, given:

```
(DEFMACRO FOO (A B) (LIST 'PLUS A B B))
```

The macro call (FOO X (BAR Y Z)) will expand to (PLUS X (BAR Y Z) (BAR Y Z)).

"&-keywords" (beginning with the character "&") that are used to set variables to particular items from the macro call form, as follows:

&OPTIONAL Used to define optional arguments, possibly with default values. Each element on *ARGS* after &OPTIONAL until the next &-keyword or the end of the list defines an optional argument, which can either be a symbol or a list, interpreted as follows:

VAR

If an optional argument is specified as a symbol, that variable is set to the corresponding element of the macro call (unevaluated).

(*VAR DEFAULT*)

If an optional argument is specified as a two element list, *VAR* is the variable to be set, and *DEFAULT* is a form that is evaluated and used as the default if there is no corresponding element in the macro call.

(*VAR DEFAULT VARSETP*)

If an optional argument is specified as a three element list, *VAR* and *DEFAULT* are the variable to be set and the default form, and *VARSETP* is a variable that is set to T if the optional argument is given in the macro call, NIL otherwise. This can be used to determine whether the argument was not given, or whether it was specified with the default value.

For example, after (DEFMACRO FOO (&OPTIONAL A (B 5) (C 6 CSET)) FORM) expanding the macro call (FOO) would cause FORM to be evaluated with A set to NIL, B set to 5, C set to 6, and CSET set to NIL. (FOO 4 5 6) would be the same, except that A would be set to 4 and CSET would be set to T.

&REST
&BODY Used to get a list of all additional arguments from the macro call. Either &REST or &BODY should be followed by a single symbol, which is set to a list of all arguments to the macro after the position of the &-keyword. For example, given

```
(DEFMACRO FOO (A B &REST C) FORM)
```

expanding the macro call (FOO 1 2 3 4 5) would cause FORM to be evaluated with A set to 1, B set to 2, and C set to (3 4 5).

If the macro calling form contains keyword arguments (see &KEY below), these are included in the &REST list.

&KEY Used to define keyword arguments, that are specified in the macro call by including a “keyword” (a symbol starting with the character “:”) followed by a value.

Each element on ARGS after &KEY until the next &-keyword or the end of the list defines a keyword argument, which can either be a symbol or a list, interpreted as follows:

VAR
(*VAR*)
((*KEYWORD VAR*))

If a keyword argument is specified by a single symbol *VAR*, or a one-element list containing *VAR*, it is set to the value of a keyword argument, where the keyword used is created by adding the character “:” to the front of *VAR*. If a keyword argument is specified by a single-element list containing a two-element list, *KEYWORD* is interpreted as the keyword (which should start with the letter “:”), and *VAR* is the variable to set.

(*VAR DEFAULT*)
((*KEYWORD VAR*) *DEFAULT*)

(*VAR DEFAULT VARSETP*)
((*KEYWORD VAR*) *DEFAULT VARSETP*)

If a keyword argument is specified by a two- or three-element list, the first element of the list specifies the keyword and variable to set as above. Similar to &OPTIONAL (above), the second element *DEFAULT* is a form that is evaluated and used as the default if there is no corresponding element in the macro call, and the third element *VARSETP* is a variable that is set to T if the optional argument is given in the macro call, NIL otherwise.

For example, the form

```
(DEFMACRO FOO (&KEY A (B 5 BSET) ((:BAR C) 6 CSET)) FORM)
```

Defines a macro with keys :A, :B (defaulting to 5), and :BAR. Expanding the macro call (FOO :BAR 2 :A 1) would cause FORM to be evaluated with A set to 1, B set to 5, BSET set to NIL, C set to 2, and CSET set to T.

&ALLOW-OTHER-KEYS It is an error for any keywords to be supplied in a macro call that are not defined as keywords in the macro argument list, unless either the &-keyword &ALLOW-OTHER-KEYS appears in *ARGS*, or the keyword :ALLOW-OTHER-KEYS (with a non-NIL value) appears in the macro call.

&AUX Used to bind and initialize auxiliary varables, using a syntax similar to PROG (see the PROG and Associated Control Functions section of Chapter 9). Any elements after &AUX should be either symbols or lists, interpreted as follows:

VAR

Single symbols are interpreted as auxiliary variables that are initially bound to NIL.

(*VAR EXP*)

If an auxiliary variable is specified as a two element list, *VAR* is a variable initially bound to the result of evaluating the form *EXP*.

For example, given

```
(DEFMACRO FOO (A B &AUX C (D 5)) FORM)
```

C will be bound to NIL and D to 5 when *FORM* is evaluated.

&WHOLE Used to get the whole macro calling form. Should be the first element of *ARGS*, and should be followed by a single symbol, which is set to the entire macro calling form. Other &-keywords or arguments can follow. For example, given

```
(DEFMACRO FOO (&WHOLE X A B) FORM)
```

Expanding the macro call (FOO 1 2) would cause FORM to be evaluated with X set to (FOO 1 2), A set to 1, and B set to 2.

DEFMACRO macros also support argument list "destructuring," a facility for accessing the structure of individual arguments to a macro. Any place in an argument list where a symbol is expected, an argument list (in the form described above) can appear instead. Such an embedded argument list is used to match the corresponding parts of that particular argument, which should be a list structure in the same form. In the simplest case, where the embedded argument list does not include &-keywords, this provides a simple way of picking apart list structures passed as arguments to a macro. For example, given

```
(DEFMACRO FOO (A (B (C . D)) E) FORM)
```

Expanding the macro call (FOO 1 (2 (3 4 5)) 6) would cause FORM to be evaluated with with A set to 1, B set to 2, C set to 3, D set to (4 5), and E set to 6. Note that the embedded argument list (B (C . D)) has an embedded argument list (C . D). Also notice that if an argument list ends in a dotted pair, that the final symbol matches the rest of the arguments in the macro call.

An embedded argument list can also include &-keywords, for interpreting parts of embedded list structures as if they appeared in a top-level macro call. For example, given

```
(DEFMACRO FOO (A (B &OPTIONAL (C 6)) D) FORM)
```

Expanding the macro call (FOO 1 (2) 3) would cause FORM to be evaluated with with A set to 1, B set to 2, C set to 6 (because of the default value), and D set to 3.

Warning: Embedded argument lists can only appear in positions in an argument list where a list is otherwise not accepted. In the above example, it would not be possible to specify an embedded argument list after the &OPTIONAL keyword, because it would be interpreted as an optional argument specification (with variable name, default value, set variable). However, it would be possible to specify an embedded argument list as the first element of an optional argument specification list, as so:

```
(DEFMACRO FOO (A (B &OPTIONAL ((X (Y) Z)
     '(1 (2) 3))) D) FORM)
```

In this case, X, Y, and Z default to 1, 2, and 3, respectively. Note that the "default" value has to be an appropriate list structure. Also, in this case either the whole structure (X (Y) Z) can be supplied, or it can be defaulted (i.e., is not possible to specify X while letting Y default).

Interpreting Macros

When the interpreter encounters a form CAR of which is an undefined function, it tries interpreting it as a macro. If CAR of the form has a macro definition, the macro is expanded, and the result of this expansion is evaluated in place of the original form. CLISPTRAN (see the Miscellaneous Functions and Variables section of Chapter 21) is used to save the result of this expansion so that the expansion only has to be done once. On subsequent occasions, the translation (expansion) is retrieved from CLISPARRAY the same as for other CLISP constructs.

Note: Because of the way that the evaluator processes macros, if you have a macro on FOO, then typing (FOO 'A 'B) will work, but FOO(A B) will not work.

11. VARIABLE BINDINGS AND THE STACK

Medley uses "deep binding." Every time a function is entered, a basic frame containing the new variables is put on top of the stack. Therefore, any variable reference requires searching the stack for the first instance of that variable, which makes free variable use somewhat more expensive than in a shallow binding scheme. On the other hand, spaghetti stack operations are considerably faster. Some other tricks involving copying freely-referenced variables to higher frames on the stack are also used to speed up the search.

The basic frames are allocated on a stack; for most user purposes, these frames should be thought of as containing the variable names associated with the function call, and the *current* values for that frame. The descriptions of the stack functions in below are presented from this viewpoint. Both interpreted and compiled functions store both the names and values of variables so that interpreted and compiled functions are compatible and can be freely intermixed, i.e., free variables can be used with no `SPECVAR` declarations necessary. However, it is possible to *suppress* storing of names in compiled functions, either for efficiency or to avoid a clash, via a `LOCALVAR` declaration (see the Local Variables and Special Variables section of Chapter 18). The names are also very useful in debugging, for they make possible a complete symbolic backtrace in case of error.

In addition to the binding information, additional information is associated with each function call: access information indicating the path to search the basic frames for variable bindings, control information, and temporary results are also stored on the stack in a block called the frame extension. The interpreter also stores information about partially evaluated expressions as described in the Stack and Interpreter section of Chapter 11.

Spaghetti Stack

The Bobrow/Wegbreit paper, "A Model and Stack Implementation for Multiple Environments" (*Communications of the ACM*, Vol. 16, 10, October 1973.), describes an access and control mechanism more general than a simple linear stack. The access and control mechanism used by Interlisp is a slightly modified version of the one proposed by Bobrow and Wegbreit. This mechanism is called the "spaghetti stack."

The spaghetti system presents the access and control stack as a data structure composed of "frames." The functions described below operate on this structure. These primitives allow user functions to manipulate the stack in a machine independent way. Backtracking, coroutines, and more sophisticated control schemes can be easily implemented with these primitives.

The evaluation of a function requires the allocation of storage to hold the values of its local variables during the computation. In addition to variable bindings, an activation of a function requires a return link (indicating where control is to go after the completion of the computation) and room for temporaries needed during the computation. In the spaghetti system, one "stack" is used for storing all this information, but it is best to view this stack as a tree of linked objects called frame extensions (or simply frames).

A frame extension is a variable sized block of storage containing a frame name, a pointer to some variable bindings (the `BLINK`), and two pointers to other frame extensions (the `ALINK` and `CLINK`). In addition to these components, a frame extension contains other information (such as temporaries and reference counts) that does not interest us here.

The block of storage holding the variable bindings is called a basic frame. A basic frame is essentially an array of pairs, each of which contains a variable name and its value. The reason frame extensions point to basic frames (rather than just having them “built in”) is so that two frame extensions can share a common basic frame. This allows two processes to communicate via shared variable bindings.

The chain of frame extensions which can be reached via the successive `ALINK`s from a given frame is called the “access chain” of the frame. The first frame in the access chain is the starting frame. The chain through successive `CLINK`s is called the “control chain”.

A frame extension completely specifies the variable bindings and control information necessary for the evaluation of a function. Whenever a function (or in fact, any form which generally binds local variables) is evaluated, it is associated with some frame extension.

In the beginning there is precisely one frame extension in existence. This is the frame in which the top-level call to the interpreter is being run. This frame is called the “top-level” frame.

Since precisely one function is being executed at any instant, exactly one frame is distinguished as having the “control bubble” in it. This frame is called the active frame. Initially, the top-level frame is the active frame. If the computation in the active frame invokes another function, a new basic frame and frame extension are built. The frame name of this basic frame will be the name of the function being called. The `ALINK`, `BLINK`, and `CLINK` of the new frame all depend on precisely how the function is invoked. The new function is then run in this new frame by passing control to that frame, i.e., it is made the active frame.

Once the active computation has been completed, control normally returns to the frame pointed to by the `CLINK` of the active frame. That is, the frame in the `CLINK` becomes the active frame.

In most cases, the storage associated with the basic frame and frame extension just abandoned can be reclaimed. However, it is possible to obtain a pointer to a frame extension and to “hold on” to this frame even after it has been exited. This pointer can be used later to run another computation in that environment, or even “continue” the exited computation.

A separate data type, called a stack pointer, is used for this purpose. A stack pointer is just a cell that literally points to a frame extension. Stack pointers print as `#ADR/FRAMENAME`, e.g., `#1,13636/COND`. Stack pointers are returned by many of the stack manipulating functions described below. Except for certain abbreviations (such as “the frame with such-and-such a name”), stack pointers are the only way you can reference a frame extension. As long as you have a stack pointer which references a frame extension, that frame extension (and all those that can be reached from it) will not be garbage collected.

Two stack pointers referencing the same frame extension are *not* necessarily `EQ`, i.e., `(EQ (STKPOS 'FOO) (STKPOS 'FOO)) = NIL`. However, `EQP` can be used to test if two different stack pointers reference the same frame extension (see the Equality Predicates section of Chapter 9).

It is possible to evaluate a form with respect to an access chain other than the current one by using a stack pointer to refer to the head of the access chain desired. Note, however, that this can be very expensive when using a shallow binding scheme such as that in Interlisp-10. When evaluating the form, since all references to variables under the shallow binding scheme go through the variable’s value cell, the values in the value cells must be adjusted to reflect the values appropriate to the desired access chain. This is done by changing all the bindings on the current access chain (all the name-value pairs) so that they contain the value current at the time of the call. Then along the new access path, all bindings are made to contain the previous value of the variable, and the current value is placed in the value cell. For that part of the access path which is shared by the old and new chain, no work has to be done. The context switching time, i.e. the overhead in switching from the current, active, access chain to another one, is directly proportional to the size of the two branches that are not shared between the access contexts. This cost should be remembered in using generators and coroutines (see the Generators section below).

Stack Functions

In the descriptions of the stack functions below, when we refer to an argument as a stack descriptor, we mean that it is one of the following:

A stack pointer — An object that points to a frame on the stack. Stack pointers are returned by many of the stack manipulating functions described below.

`NIL` — Specifies the active frame; that is, the frame of the stack function itself.

`T` — Specifies the top-level frame.

A symbol — Specifies the first frame (along the control chain from the active frame) that has the frame name `LITATOM`. Equivalent to `(STKPOS LITATOM -1)`.

A list of symbols — Specifies the first frame (along the control chain from the active frame) whose frame name is included in the list.

A number *N* — Specifies the *N*th frame back from the active frame. If *N* is negative, the control chain is followed, otherwise the access chain is followed. Equivalent to `(STKNTH N)`.

In the stack functions described below, the following errors can occur: The error `Illegal stack arg` occurs when a stack descriptor is expected and the supplied argument is either not a legal stack descriptor (i.e., not a stack pointer, symbol, or number), or is a symbol or number for which there is no corresponding stack frame, e.g., `(STKNTH -1 'FOO)` where there is no frame named `FOO` in the active control chain or `(STKNTH -10 'EVALQT)`. The error `Stack pointer has been released` occurs whenever a released stack pointer is supplied as a stack descriptor argument for any purpose other than as a stack pointer to re-use.

Note: The creation of a single stack pointer can result in the retention of a large amount of stack space. Therefore, one should try to release stack pointers when they are no longer needed (see the Releasing and Reusing Stack Pointers section below).

In Lisp there is a fixed ammount of space allocated for the stack. When most of this space is exhausted, the `STACK OVERFLOW` error occurs and the debugger will be invoked. You will still have a little room on the stack to use inside the debugger. If you use up this last little bit of stack you will encounter a “hard” stack overflow. A “hard” stack overflow will put you into URaid (see the documentation on URaid).

Searching the Stack

(**STKPOS** *FRAMENAME N POS OLDPOS*) [Function]

Returns a stack pointer to the *N*th frame with frame name *FRAMENAME*. The search begins with (and includes) the frame specified by the stack descriptor *POS*. The search proceeds along the control chain from *POS* if *N* is negative, or along the access chain if *N* is positive. If *N* is NIL, -1 is used. Returns a stack pointer to the frame if such a frame exists, otherwise returns NIL. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is supplied and is a stack pointer and STKPOS returns NIL, *OLDPOS* is released. If *OLDPOS* is not a stack pointer it is ignored.

(**STKNTH** *N POS OLDPOS*) [Function]

Returns a stack pointer to the *N*th frame back from the frame specified by the stack descriptor *POS*. If *N* is negative, the control chain from *POS* is followed. If *N* is positive the access chain is followed. If *N* equals 0, STKNTH returns a stack pointer to *POS* (this provides a way to copy a stack pointer). Returns NIL if there are fewer than *N* frames in the appropriate chain. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is not a stack pointer it is ignored.

Note: (STKNTH 0) causes an error, Illegal stack arg; it is not possible to create a stack pointer to the active frame.

(**STKNAME** *POS*) [Function]

Returns the frame name of the frame specified by the stack descriptor *POS*.

(**SETSTKNAME** *POS NAME*) [Function]

Changes the frame name of the frame specified by *POS* to be *NAME*. Returns *NAME*.

(**STKNTHNAME** *N POS*) [Function]

Returns the frame name of the *N*th frame back from *POS*. Equivalent to (STKNAME (STKNTH *N POS*)) but avoids creation of a stack pointer.

In summary, STKPOS converts function names to stack pointers, STKNTH converts numbers to stack pointers, STKNAME converts stack pointers to function names, and STKNTHNAME converts numbers to function names.

Variable Bindings in Stack Frames

The following functions are used for accessing and changing bindings. Some of functions take an argument, *N*, which specifies a particular binding in the basic frame. If *N* is a literal atom, it is assumed to be the name of a variable bound in the basic frame. If *N* is a number, it is assumed to reference the *N*th binding in the basic frame. The first binding is 1. If the basic frame contains no binding with the given name or if the number is too large or too small, the error Illegal arg occurs.

(**STKSCAN** *VAR IPOS OPOS*) [Function]

Searches beginning at *IPOS* for a frame in which a variable named *VAR* is bound. The search follows the access chain. Returns a stack pointer to the frame if found, otherwise returns NIL. If *OPOS* is a stack pointer it is reused, otherwise it is ignored.

(**FRAMESCAN** *ATOM POS*) [Function]

Returns the relative position of the binding of *ATOM* in the basic frame of *POS*. Returns NIL if *ATOM* is not found.

(**STKARG** *N POS* —) [Function]

Returns the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(**STKARGNAME** *N POS*) [Function]

Returns the name of the binding specified by *N*, in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(**SETSTKARG** *N POS VAL*) [Function]

Sets the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *VAL*.

(**SETSTKARGNAME** *N POS NAME*) [Function]

Sets the variable name to *NAME* of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *NAME*. This function does not work for interpreted frames.

(**STKNARGS** *POS* —) [Function]

Returns the number of arguments bound in the basic frame of the frame specified by the stack descriptor *POS*.

(**VARIABLES** *POS*) [Function]

Returns a list of the variables bound at *POS*.

(**STKARGS** *POS* —) [Function]

Returns a list of the values of the variables bound at *POS*.

Evaluating Expressions in Stack Frames

The following functions are used to evaluate an expression in a different environment:

(**ENVEVAL** *FORM APOS CPOS AFLG CFLG*) [Function]

Evaluates *FORM* in the environment specified by *APOS* and *CPOS*. That is, a new active frame is created with the frame specified by the stack descriptor *APOS* as its ALINK, and the frame specified by the stack descriptor *CPOS* as its CLINK. Then *FORM* is evaluated. If

AFLG is not NIL, and *APOS* is a stack pointer, then *APOS* will be released. Similarly, if *CFLG* is not NIL, and *CPOS* is a stack pointer, then *CPOS* will be released.

(**ENVAPPLY** *FN ARGS APOS CPOS AFLG CFLG*) [Function]

APPLYs *FN* to *ARGS* in the environment specified by *APOS* and *CPOS*. *AFLG* and *CFLG* have the same interpretation as with ENVEVAL.

(**EVALV** *VAR POS RELFLG*) [Function]

Evaluates *VAR*, where *VAR* is assumed to be a symbol, in the access environment specifed by the stack descriptor *POS*. If *VAR* is unbound, EVALV returns NOBIND and does not generate an error. If *RELFLG* is non-NIL and *POS* is a stack pointer, it will be released after the variable is looked up. While EVALV could be defined as (ENVEVAL *VAR POS* NIL *RELFLG*) it is in fact somewhat faster.

(**STKEVAL** *POS FORM FLG* —) [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*. If *FLG* is not NIL and *POS* is a stack pointer, releases *POS*. The definition of STKEVAL is (ENVEVAL *FORM POS* NIL *FLG*).

(**STKAPPLY** *POS FN ARGS FLG*) [Function]

Like STKEVAL but applies *FN* to *ARGS*.

Altering Flow of Control

The following functions are used to alter the normal flow of control, possibly jumping to a different frame on the stack. RETEVAL and RETAPPLY allow evaluating an expression in the specified environment first.

(**RETFROM** *POS VAL FLG*) [Function]

Return from the frame specified by the stack descriptor *POS*, with the value *VAL*. If *FLG* is not NIL, and *POS* is a stack pointer, then *POS* is released. An attempt to RETFROM the top level (e.g., (RETFROM T)) causes an error, Illegal stack arg. RETFROM can be written in terms of ENVEVAL as follows:

```
(RETFROM
     (LAMBDA (POS VAL FLG)
        (ENVEVAL (LIST 'QUOTE VAL)
                 NIL
                 (if (STKNTH -1 POS
                     (if FLG then POS))
                 else (ERRORX (LIST 19 POS)))
                 NIL
                 T)))
```

(**RETTO** *POS VAL FLG*) [Function]

Like RETFROM, but returns *to* the frame specified by *POS*.

(**RETEVAL** *POS FORM FLG* —) [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*, and then returns from *POS* with that value. If *FLG* is not NIL and *POS* is a stack pointer, then *POS* is released. The definition of RETEVAL is equivalent to (ENVEVAL *FORM POS* (STKNTH -1 *POS*) *FLG* T), but RETEVAL does not create a stack pointer.

(**RETAPPLY** *POS FN ARGS FLG*) [Function]

Like RETEVAL but applies *FN* to *ARGS*.

Releasing and Reusing Stack Pointers

The following functions and variables are used for manipulating stack pointers:

(**STACKP** *X*) [Function]

Returns *X* if *X* is a stack pointer, otherwise returns NIL.

(**RELSTK** *POS*) [Function]

Release the stack pointer *POS* (see below). If *POS* is not a stack pointer, does nothing. Returns *POS*.

(**RELSTKP** *X*) [Function]

Returns T is *X* is a released stack pointer, NIL otherwise.

(**CLEARSTK** *FLG*) [Function]

If *FLG* is T, returns a list of all the active (unreleased) stack pointers. If *FLG* is NIL, this call is a no-op. The abillity to clear all stack pointers is inconsistent with the modularity implicit in a multi processing environment.

CLEARSTKLST [Variable]

A variable used by the top-level executive. Every time the top-level executive is re-entered (e.g., following errors, or Control-D), CLEARSTKLST is checked. If its value is T, all active stack pointers are released using CLEARSTK. If its value is a list, then all stack pointers on that list are released. If its value is NIL, nothing is released. CLEARSTKLST is initially T.

NOCLEARSTKLST [Variable]

A variable used by the top-level executive. If CLEARSTKLST is T (see above) all active stack pointers *except* those on NOCLEARSTKLST are released. NOCLEARSTKLST is initially NIL.

Creating a single stack pointer can cause the retention of a large amount of stack space. Furthermore, this space will not be freed until the next garbage collection, *even if the stack pointer is no longer being used*, unless the stack pointer is explicitly released or reused. If there is sufficient amount of stack space tied up in this fashion, a STACK OVERFLOW condition can occur, even in the simplest of computations. For this reason, you should consider releasing a stack pointer when the environment referenced by the stack pointer is no longer needed.

The effects of releasing a stack pointer are:

1. The link between the stack pointer and the stack is broken by setting the contents of the stack pointer to the "released mark". A released stack pointer prints as #*ADR*/#0.

2. If this stack pointer was the last remaining reference to a frame extension; that is, if no other stack pointer references the frame extension and the extension is not contained in the active control or access chain, then the extension may be reclaimed, and is reclaimed immediately. The process repeats for the access and control chains of the reclaimed extension so that all stack space that was reachable only from the released stack pointer is reclaimed.

A stack pointer may be released using the function RELSTK, but there are some cases for which RELSTK is not sufficient. For example, if a function contains a call to RETFROM in which a stack pointer was used to specify where to return to, it would not be possible to simultaneously release the stack pointer. (A RELSTK appearing in the function following the call to RETFROM would not be executed!) To permit release of a stack pointer in this situation, the stack functions that relinquish control have optional flag arguments to denote whether or not a stack pointer is to be released (*AFLG* and *CFLG*). Note that in this case releasing the stack pointer will *not* cause the stack space to be reclaimed immediately because the frame referenced by the stack pointer will have become part of the active environment.

Another way to avoid creating new stack pointers is to *reuse* stack pointers that are no longer needed. The stack functions that create stack pointers (STKPOS, STKNTH, and STKSCAN) have an optional argument that is a stack pointer to reuse. When a stack pointer is reused, two things happen. First the stack pointer is released (see above). Then the pointer to the new frame extension is deposited in the stack pointer. The old stack pointer (with its new contents) is returned as the value of the function. Note that the reused stack pointer will be released even if the function does not find the specified frame.

Even if stack pointers are explicitly being released, *creating* many stack pointers can cause a garbage collection of stack pointer space. Thus, if your application requires creating many stack pointers, you definitely should take advantage of reusing stack pointers.

Backtrace Functions

The following functions perform a "backtrace," printing information about every frame on the stack. Arguments allow only backtracing a selected range of the stack, skipping selected frames, and printing different amounts of information about each frame.

(**BACKTRACE** *IPOS EPOS FLAGS FILE PRINTFN*) [Function]

Performs a backtrace beginning at the frame specified by the stack descriptor *IPOS*, and ending with the frame specified by the stack descriptor *EPOS*. *FLAGS* is a number in which the options of the BACKTRACE are encoded. If a bit is set, the corresponding information is included in the backtrace.

1Q - print arguments of non-SUBRs

2Q - print temporaries of the interpreter

4Q - print SUBR arguments and local variables

10Q - omit printing of UNTRACE: and function names

20Q - follow access chain instead of control chain

40Q - print temporaries, i.e. the blips (see the stack and interpreter section below)

For example: If *FLAGS* = 47Q, everything is printed. If *FLAGS* = 21Q, follows the access chain, prints arguments.

FILE is the file that the backtrace is printed to. *FILE* must be open. *PRINTFN* is used when printing the values of variables, temporaries, blips, etc. *PRINTFN* = NIL defaults to PRINT.

(**BAKTRACE** *IPOS EPOS SKIPFNS FLAGS FILE*) [Function]

Prints a backtrace from *IPOS* to *EPOS* onto *FILE*. *FLAGS* specifies the options of the backtrace, e.g., do/don't print arguments, do/don't print temporaries of the interpreter, etc., and is the same as for BACKTRACE.

SKIPFNS is a list of functions. As BAKTRACE scans down the stack, the stack name of each frame is passed to each function in *SKIPFNS*, and if any of them returnS non-NIL, *POS* is skipped (including all variables).

BAKTRACE collapses the sequence of several function calls corresponding to a call to a system package into a single "function" using BAKTRACELST as described below. For example, any call to the editor is printed as **EDITOR**, a break is printed as **BREAK**, etc.

BAKTRACE is used by the BT, BTV, BTV+, BTV*, and BTV! break commands, with *FLAGS* = 0, 1, 5, 7, and 47Q respectively.

If SYSPRETTYFLG = T, the values arguments and local variables will be prettyprinted.

BAKTRACELST [Variable]

Used to tell BAKTRACE (therefore, the BT, BTV, etc. commands) to abbreviate various sequences of function calls on the stack by a single key, e.g. **BREAK**, **EDITOR**, etc.

Each entry on BAKTRACELST is a list of the form (*FRAMENAME KEY* . *PATTERN*) or (*FRAMENAME* (KEY_1 . $PATTERN_1$) ... (KEY_N . $PATTERN_N$)), where a pattern is a list of elements that are either atoms, which match a single frame, or lists, which are interpreted as a list of alternative patterns, e.g. (PROGN **BREAK** EVAL ((ERRORSET BREAK1A BREAK1) (BREAK1)))

BAKTRACE operates by scanning up the stack and, at each point, comparing the current frame name, with the frame names on BAKTRACELST, i.e. it does an ASSOC. If the frame name does appear, BAKTRACE attempts to match the stack as of that point with (one of) the patterns. If the match is successful, BAKTRACE prints the corresponding key, and continues with where the match left off. If the frame name does not appear, or the match fails, BAKTRACE simply prints the frame name and continues with the next higher frame (unless the *SKIPFNS* applied to the frame name are non-NIL as described above).

Matching is performed by comparing symbols in the pattern with the current frame name, and matching lists as patterns, i.e. sequences of function calls, always working up the stack. For example, either of the sequence of function calls "... BREAK1 BREAK1A ERRORSET EVAL PROGN ..."

or "... BREAK1 EVAL PROGN ..." would match with the sample entry given above, causing **BREAK** to be printed.

Special features:

- The symbol & can be used to match any frame.
- The pattern "-" can be used to match nothing. - is useful for specifying an optional match, e.g. the example above could also have been written as (PROGN **BREAK** EVAL ((ERRORSET BREAK1A) -) BREAK1).
- It is not necessary to provide in the pattern for matching dummy frames, i.e. frames for which DUMMYFRAMEP (see below) is true. When working on a match, the matcher automatically skips over these frames when they do not match.
- If a match succeeds and the *KEY* is NIL, nothing is printed. For example, (*PROG*LAM NIL EVALA *ENV). This sequence will occur following an error which then causes a break if some of the function's arguments are LOCALVARS.

Other Stack Functions

(**DUMMYFRAMEP** *POS*) [Function]

Returns T if you never wrote a call to the function at *POS*, e.g. in Interlisp-10, DUMMYFRAMEP is T for *PROG*LAM, *ENV*, and FOOBLOCK frames (see the Block Compiling section of Chapter 18).

REALFRAMEP and REALSTKNTH can be used to write functions which manipulate the stack and work on either interpreted or compiled code:

(**REALFRAMEP** *POS INTERPFLG*) [Function]

Returns *POS* if *POS* is a "real" frame, i.e. if *POS* is not a dummy frame and *POS* is a frame that does not disappear when compiled (such as COND); otherwise NIL. If *INTERPFLG* = T, returns T if *POS* is not a dummy frame. For example, if (STKNAME *POS*) = COND, (REALFRAMEP *POS*) is NIL, but (REALFRAMEP *POS* T) is T.

(**REALSTKNTH** *N POS INTERPFLG OLDPOS*) [Function]

Returns a stack pointer to the *N*th (or -*N*th) frames for which (REALFRAMEP *POS INTERPFLG*) is *POS*.

(**MAPDL** *MAPDLFN MAPDLPOS*) [Function]

Starts at *MAPDLPOS* and applies the function *MAPDLFN* to two arguments (the frame name and a stack pointer to the frame), for each frame until the top of the stack is reached. Returns NIL. For example,

```
[MAPDL (FUNCTION (LAMBDA (X POS)
   (if (IGREATERP (STKNARGS POS) 2) then (PRINT X)]
```

will print all functions of more than two arguments.

(**SEARCHPDL** *SRCHFN SRCHPOS*) [Function]

Like MAPDL, but searches the stack starting at position *SRCHPOS* until it finds a frame for which *SRCHFN*, a function of two arguments applied to the *name* of the frame and the frame itself, is not NIL. Returns (*NAME* . *FRAME*) if such a frame is found, otherwise NIL.

The Stack and the Interpreter

In addition to the names and values of arguments for functions, information regarding partially-evaluated expressions is kept on the push-down list. For example, consider the following definition of the function FACT (intentionally faulty):

```
(FACT
   [LAMBDA (N)
      (COND
        ((ZEROP N)
          L)
        (T (ITIMES N (FACT (SUB1 N])
```

In evaluating the form (FACT 1), as soon as FACT is entered, the interpreter begins evaluating the implicit PROGN following the LAMBDA. The first function entered in this process is COND. COND begins to process its list of clauses. After calling ZEROP and getting a NIL value, COND proceeds to the next clause and evaluates T. Since T is true, the evaluation of the implicit PROGN that is the consequent of the T clause is begun. This requires calling the function ITIMES. However before ITIMES can be called, its arguments must be evaluated. The first argument is evaluated by retrieving the current binding of N from its value cell; the second involves a recursive call to FACT, and another implicit PROGN, etc.

At each stage of this process, some portion of an expression has been evaluated, and another is awaiting evaluation. The output below (from Interlisp-10) illustrates this by showing the state of the push-down list at the point in the computation of (FACT 1) when the unbound atom L is reached.

```
← FACT(1)
 u.b.a. L {in FACT} in ((ZEROP NO L)
 (L broken)
 :BTV!
    *TAIL* (L)
    *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
 COND
    *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
    *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
    N 0
 FACT
    *FORM* (FACT (SUB1 N))
    *FN* ITIMES
    *TAIL* ((FACT (SUB1 N)))
    *ARGVAL* 1
    *FORM* (ITIMES N (FACT (SUB1 N)))
    *TAIL* ((ITIMES N (FACT (SUB1 N))))
    *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
 COND
    *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
    *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
    N 1
 FACT
```

```
**TOP**
```

Internal calls to EVAL, e.g., from COND and the interpreter, are marked on the push-down list by a special mark or blip which the backtrace prints as *FORM*. The genealogy of *FORM*'s is thus a history of the computation. Other temporary information stored on the stack by the interpreter includes the tail of a partially evaluated implicit PROGN (e.g., a cond clause or lambda expression) and the tail of a partially evaluated form (i.e., those arguments not yet evaluated), both indicated on the backtrace by *TAIL*, the values of arguments that have already been evaluated, indicated by *ARGVAL*, and the names of functions waiting to be called, indicated by *FN*. *ARG1, ..., *ARGn are used by the backtrace to indicate the (unnamed) arguments to SUBRs.

Note that a function is not actually entered and does not appear on the stack, until its arguments have been evaluated (except for nlambda functions, of course). Also note that the *ARG1, *FORM*, *TAIL*, etc. "bindings" comprise the actual working storage. In other words, in the above example, if a (lower) function changed the value of the *ARG1 binding, the COND would continue interpreting the new binding as a list of COND clauses. Similarly, if the *ARGVAL* binding were changed, the new value would be given to ITIMES as its first argument after its second argument had been evaluated, and ITIMES was actually called.

FORM, *TAIL*, *ARGVAL*, etc., do not actually appear as variables on the stack, i.e., evaluating *FORM* or calling STKSCAN to search for it will not work. However, the functions BLIPVAL, SETBLIPVAL, and BLIPSCAN described below are available for accessing these internal blips. These functions currently know about four different types of blips:

FN	The name of a function about to be called
ARGVAL	An argument for a function about to be called
FORM	A form in the process of evaluation
TAIL	The tail of a COND clause, implicit PROGN, PROG, etc.

(**BLIPVAL** *BLIPTYP IPOS FLG*) [Function]

Returns the value of the specified blip of type *BLIPTYP*. If *FLG* is a number N, finds the Nth blip of the desired type, searching the control chain beginning at the frame specified by the stack descriptor *IPOS*. If *FLG* is NIL, 1 is used. If *FLG* is T, returns the number of blips of the specified type at *IPOS*.

(**SETBLIPVAL** *BLIPTYP IPOS N VAL*) [Function]

Sets the value of the specified blip of type *BLIPTYP*. Searches for the *N*th blip of the desired type, beginning with the frame specified by the stack descriptor *IPOS*, and following the control chain.

(**BLIPSCAN** *BLIPTYP IPOS*) [Function]

Returns a stack pointer to the frame in which a blip of type *BLIPTYP* is located. Search begins at the frame specified by the stack descriptor *IPOS* and follows the control chain.

Generators

A *generator* is like a subroutine except that it retains information about previous times it has been called. Some of this state may be data (for example, the seed in a random number generator), and some may be in program state (as in a recursive generator which finds all the atoms in a list structure). For example, if LISTGEN is defined by:

```
(DEFINEQ (LISTGEN (L)
        (if L then (PRODUCE (CAR L))
                   (LISTGEN (CDR L))))
```

we can use the function GENERATOR (described below) to create a generator that uses LISTGEN to produce the elements of a list one at a time, e.g.,

```
(SETQ GR (GENERATOR (LISTGEN '(A B C))))
```

creates a generator, which can be called by

```
(GENERATE GR)
```

to produce as values on successive calls, A, B, C. When GENERATE (not GENERATOR) is called the first time, it simply starts evaluating (LISTGEN '(A B C)). PRODUCE gets called from LISTGEN, and pops back up to GENERATE with the indicated value after saving the state. When GENERATE gets called again, it continues from where the last PRODUCE left off. This process continues until finally LISTGEN completes and returns a value (it doesn't matter what it is). GENERATE then returns GR itself as its value, so that the program that called GENERATE can tell that it is finished, i.e., there are no more values to be generated.

(**GENERATOR** *FORM COMVAR*) [NLambda Function]

An nlambda function that creates a generator which uses *FORM* to compute values. GENERATOR returns a *generator handle* which is represented by a dotted pair of stack pointers.

COMVAR is optional. If its value (EVAL of) is a generator handle, the list structure and stack pointers will be reused. Otherwise, a new generator handle will be constructed.

GENERATOR compiles open.

(**PRODUCE** *VAL*) [Function]

Used from within a generator to return *VAL* as the value of the corresponding call to GENERATE.

(**GENERATE** *HANDLE VAL*) [Function]

Restarts the generator represented by *HANDLE*. *VAL* is returned as the value of the PRODUCE which last suspended the operation of the generator. When the generator runs out of values, GENERATE returns *HANDLE* itself.

Examples:

The following function will go down recursively through a list structure and produce the atoms in the list structure one at a time.

```
(DEFINEQ (LEAVESG (L)
       (if (ATOM L)
```

```
              then (PRODUCE L)
              else (LEAVESG (CAR L))
                   (if (CDR L)
                    then (LEAVESG (CDR L)]
```

The following function prints each of these atoms as it appears. It illustrates how a loop can be set up to use a generator.

```
(DEFINEQ (PLEAVESG1 (L)
          (PROG (X LHANDLE)
             (SETQ LHANDLE (GENERATOR (LEAVESG L)))
              LP (SETQ X (GENERATE LHANDLE))
                (if (EQ X LHANDLE)
                 then (RETURN NIL))
                  (PRINT X)
                     (GO LP))]
```

The loop terminates when the value of the generator is EQ to the dotted pair which is the value produced by the call to GENERATOR. A CLISP iterative operator, OUTOF, is provided which makes it much easier to write the loop in PLEAVESG1. OUTOF (or outof) can precede a form which is to be used as a generator. On each iteration, the iteration variable will be set to successive values returned by the generator; the loop will be terminated automatically when the generator runs out. Therefore, the following is equivalent to the above program PLEAVESG1:

```
(DEFINEQ (PLEAVESG2 (L) (for X outof (LEAVESG L) do (PRINT X))]
```

Here is another example; the following form will print the first N atoms.

```
(for X outof (MAPATOMS (FUNCTION PRODUCE)) as I from 1 to N do (PRINT X))
```

Coroutines

This package provides facilities for the creation and use of fully general coroutine structures. It uses a stack pointer to preserve the state of a coroutine, and allows arbitrary switching between *N* different coroutines, rather than just a call to a generator and return. This package is slightly more efficient than the generator package described above, and allows more flexibility on specification of what to do when a coroutine terminates.

(**COROUTINE** *CALLPTR COROUTPTR COROUTFORM ENDFORM*) [NLambda Function]

This nlambda function is used to create a coroutine and initialize the linkage. *CALLPTR* and *COROUTPTR* are the names of two variables, which will be set to appropriate stack pointers. If the values of *CALLPTR* or *COROUTPTR* are already stack pointers, the stack pointers will be reused. *COROUTFORM* is the form which is evaluated to start the coroutine; *ENDFORM* is a form to be evaluated if *COROUTFORM* actually returns when it runs out of values.

COROUTINE compiles open.

(**RESUME** *FROMPTR TOPTR VAL*) [Function]

Used to transfer control from one coroutine to another. *FROMPTR* should be the stack pointer for the current coroutine, which will be smashed to preserve the current state. *TOPTR* should be the stack pointer which has preserved the state of the coroutine to be transferred to, and *VAL* is the value that is to be returned to the latter coroutine as the value of the RESUME which suspended the operation of that coroutine.

For example, the following is the way one might write the LEAVES program using the coroutine package:

```
(DEFINEQ (LEAVESC (L COROUTPTR CALLPTR)
    (if (ATOM L)
     then (RESUME COROUTPTR CALLPTR L)
     else (LEAVESC (CAR L) COROUTPTR CALLPTR)
          (if (CDR L) then (LEAVESC (CDR L) COROUTPTR CALLPTR))))]
```

A function PLEAVESC which uses LEAVESC can be defined as follows:

```
(DEFINEQ (PLEAVESC (L)
  (bind PLHANDLE LHANDLE
   first (COROUTINE PLHANDLE LHANDLE
                    (LEAVESC L LHANDLE PLHANDLE)
                    (RETFROM 'PLEAVESC))
   do (PRINT (RESUME PLHANDLE LHANDLE))))]
```

By RESUMEing LEAVESC repeatedly, this function will print all the leaves of list L and then return out of PLEAVESC via the RETFROM. The RETFROM is necessary to break out of the non-terminating do-loop. This was done to illustrate the additional flexibility allowed through the use of *ENDFORM*.

We use two coroutines working on two trees in the example EQLEAVES, defined below. EQLEAVES tests to see whether two trees have the same leaf set in the same order, e.g., (EQLEAVES '(A B C) '(A B (C))) is true.

```
(DEFINEQ (EQLEAVES (L1 L2)
    (bind LHANDLE1 LHANDLE2 PE EL1 EL2
     first (COROUTINE PE LHANDLE1 (LEAVESC L1 LHANDLE1 PE) 'NO-MORE)
           (COROUTINE PE LHANDLE2 (LEAVESC L2 LHANDLE2 PE) 'NO-MORE)
     do (SETQ EL1 (RESUME PE LHANDLE1))
        (SETQ EL2 (RESUME PE LHANDLE2))
        (if (NEQ EL1 EL2)
         then (RETURN NIL))
     repeatuntil (EQ EL1 'NO-MORE)
     finally (RETURN T)))]
```

Possibilities Lists

A possibilities list is the interface between a generator and a consumer. The possibilities list is initialized by a call to POSSIBILITIES, and elements are obtained from it by using TRYNEXT. By using the spaghetti stack to maintain separate environments, this package allows a regime in which a generator can put a few items in a possibilities list, suspend itself until they have been consumed, and be subsequently aroused and generate some more.

(**POSSIBILITIES** *FORM*) [NLambda Function]

This nlambda function is used for the initial creation of a possibilities list. *FORM* will be evaluated to create the list. It should use the functions NOTE and AU-REVOIR described below to generate possibilities. Normally, one would set some variable to the possibilities list which is returned, so it can be used later, e.g.:

```
(SETQ PLIST (POSSIBILITIES (GENERFN V1 V2))).
```

POSSIBILITIES compiles open.

(**NOTE** *VAL LSTFLG*) [Function]

Used within a generator to put items on the possibilities list being generated. If *LSTFLG* is equal to NIL, *VAL* is treated as a single item. If *LSTFLG* is non-NIL, then the list *VAL* is NCONCed on the end of the possibilities list. Note that it is perfectly reasonable to create a possibilities list using a second generator, and NOTE that list as possibilities for the current generator with *LSTFLG* equal to T. The lower generator will be resumed at the appropriate point.

(**AU-REVOIR** *VAL*) [NoSpread Function]

Puts *VAL* on the possibilities list if it is given, and then suspends the generator and returns to the consumer in such a fashion that control will return to the generator at the AU-REVOIR if the consumer exhausts the possibilities list.

NIL is not put on the possibilities list unless it is explicitly given as an argument to AU-REVOIR, i.e., (AU-REVOIR) and (AU-REVOIR NIL) are *not* the same. AU-REVOIR and ADIEU are lambda nospreads to enable them to distinguish these two cases.

(**ADIEU** *VAL*) [NoSpread Function]

Like AU-REVOIR but releases the generator instead of suspending it.

(**TRYNEXT** *PLST ENDFORM VAL*) [NLambda Function]

This nlambda function allows a consumer to use a possibilities list. It removes the first item from the possibilities list named by *PLST* (i.e. *PLST* must be an atom whose value is a possiblities list), and returns that item, provided it is not a generator handle. If a generator handle is encountered, the generator is reawakened. When it returns a possibilities list, this list is added to the front of the current list. When a call to TRYNEXT causes a generator to be awakened, *VAL* is returned as the value of the AU-REVOIR which put that generator to sleep. If *PLST* is empty, it evaluates *ENDFORM* in the caller's environment.

TRYNEXT compiles open.

(**CLEANPOSLST** *PLST*) [Function]

This function is provided to release any stack pointers which may be left in the *PLST* which was not used to exhaustion.

For example, FIB is a generator for fibonnaci numbers. It starts out by NOTEing its two arguments, then suspends itself. Thereafter, on being re-awakened, it will NOTE two more terms in the series and suspends again. PRINTFIB uses FIB to print the first N fibonacci numbers.

```
(DEFINEQ (FIB (F1 F2)
  (do (NOTE F1)
      (NOTE F2)
      (SETQ F1 (IPLUS F1 F2))
      (SETQ F2 (IPLUS F1 F2))
      (AU-REVOIR)]
```

Note that this AU-REVOIR just suspends the generator and adds nothing to the possibilities list except the generator.

```
(DEFINEQ (PRINTFIB (N)
    (PROG ((FL (POSSIBILITIES (FIB 0 1))))
          (RPTQ N (PRINT  (TRYNEXT FL)))
          (CLEANPOSLST FL)]
```

Note that FIB itself will never terminate.

[This page intentionally left blank]

Greeting and Initialization Files

Many of the features of Medley are controlled by variables that you can adjust to your own taste. In addition, you can modify the action of system functions in ways not specifically provided for by using ADVISE (see the Advise Functions section of Chapter 15). To encourage customizing Medley's environment, it includes a facility for automatically loading initialization files (or "init files") when it is first started. Each user can have a separate "user init file" that customizes Medley's environment to his/her tastes. In addition, there can be a "site init file" that applies to all users at a given physical site, setting system variables that are the same for all users such as the name of the nearest printer, etc.

The process of loading init files, also known as "greeting", occurs when a Medley system created by MAKESYS (see the Saving Virtual Memory State section below) is started for the first time. The user can also explicitly invoke the greeting operation at any time via the function GREET (below). The process of greeting includes the following steps:

1. Any previous greeting operation is undone. The side effects of the greeting operation are stored on a global variable as well as on the history list, thus enabling the previous greeting to be undone even if it has dropped off of the bottom of the history list.

2. All of the items on the list PREGREETFORMS are evaluated.

3. The site init file is loaded. GREET looks for a file by the name {DSK}INIT.LISP. If this is found, it is loaded. If it is not found, the system prints `Please enter name of system init file (e.g. {server}<directory>INIT.extension):` and waits for you to type a file name, followed by a carriage return. If you just type a carriage return without typing a file name, no site init file is loaded. **Note**: The site init file is loaded with *LDFLG* set to SYSLOAD, so that no file package information is saved, and nothing is printed out.

4. The user init file is loaded. The user init file is found by using the variable USERGREETFILES (described below), which is normally set in the site init file. The user init file is loaded with normal file package settings, but under errorset protection and with PRETTYHEADER set to NIL to suppress the `File created` message.

5. All of the items on the list POSTGREETFORMS are evaluated.

6. The greeting "`Hello, ` *XXX*`.`" is printed, where *XXX* is the value of the variable FIRSTNAME (if non-NIL). The variable GREETDATES (below) can be set to modify this greeting for particular dates.

(**GREET** *NAME* —) [Function]

Performs the greeting for person whose username is *NAME* (if *NAME* = NIL, uses the login name). When Medley first starts up, it performs (GREET).

(**GREETFILENAME** *USER*) [Function]

If *USER* is T, GREETFILENAME returns the file name of the site init file. If the file name doesn't exist, you are prompted for it. Otherwise, *USER* is interpreted to be a user's system name, and GREETFILENAME returns the file name for the user init file (if it exists).

USERGREETFILES [Variable]

USERGREETFILES specifies a series of file names to try as the user init file. The value of USERGREETFILES is a list, where each element is a list of symbols. For each item in USERGREETFILES, the user name is substituted for the symbol USER and the value of COMPILE.EXT (see the Cimpiler Functions section of Chapter 18) is substituted for the symbol COM, and the symbols are packed into a single file name. The first such file that is found is the user init file.

For example, suppose that the value of USERGREETFILES was

```
(({ERIS}< USER >LISP>INIT. COM)
 ({ERIS}< USER >LISP>INIT)
 ({ERIS}< USER >INIT. COM)
 ({ERIS}< USER >INIT))
```

If the user name was JONES, and the value of COMPILE.EXT was DCOM, then this would search for the files {ERIS}<JONES>LISP>INIT.DCOM, {ERIS}<JONES>LISP>INIT, {ERIS}<JONES>INIT.DCOM, and {ERIS}<JONES>INIT.

Note: The file name "specifications" in USERGREETFILES should be fully qualified, including all host and directory information. The directory search path (the value of DIRECTORIES, see the Searching File Directories section of Chapter 24) is *not* used to find the user greet files.

GREETDATES [Variable]

The value of GREETDATES can be used to specify special greeting messages for various dates. GREETDATES is a list of elements of the form (*DATESTRING* . *STRING*), e.g. ("25-DEC" . "Merry Christmas"). The user can add entries to this list in his/her INIT.LISP file by using a ADDVARS file package command like (ADDVARS (GREETDATES ("8-FEB" . "Happy Birthday"))). On the specified date, the GREET will use the indicated salutation.

It is impossible to give a complete list of all of the variables and functions you may want to set in your init files. The default values for system variables are chosen in the hope that they will be correct for the majority of users, so many users get along with very small init files. The following describes some of the variables that users may want to reset in their init files:

Directories — The variables DIRECTORIES and LISPUSERSDIRECTORIES (see the Searching File Directories section of Chapter 24) contain lists of directories used when searching for files. LOGINHOST/DIR (see the Incomplete File Names section of Chapter 24) determines the default directory used when you call CONN with no argument.

Fonts and Printing — The variables DISPLAYFONTDIRECTORIES, DISPLAYFONTEXTENSIONS, INTERPRESSFONTDIRECTORIES, and PRESSFONTWIDTHSFILES (see the Font Files and Font Directories section of Chapter 27) must be set before fonts can be automatically loaded from files. DEFAULTPRINTINGHOST (see Chapter 29) should be set before attempting to generate hardcopy to a printer.

Network Systems — CH.DEFAULT.ORGANIZATION and CH.DEFAULT.DOMAIN (see the Name and Address Conventions section of Chapter 31) should be set to the default NS organization and domain, when using NS network communications. If CH.NET.HINT (see the Clearinghouse Functions section of Chapter 31) is set, it can reduce the amount of time spent searching for a clearinghouse.

Medley Executive — The variable PROMPT#FLG (see the Changing the Programmer's Assistant section of Chapter 13) determines whether an "event number" is printed at the beginning of every input line. The function CHANGESLICE (see the Changing the Programmer's Assistant section of Chapter 13) can be used to change the number of events that are remembered on the history list.

Copyright Notices — COPYRIGHTFLG, COPYRIGHTOWNERS, and DEFAULTCOPYRIGHTOWNER (see the Copyright Notices section of Chapter 17) control the inclusion of copyright notices on source files.

Printing Functions — **COMMENT**FLG (see the Comment Feature section of Chapter 26) determines how program comments are printed. FIRSTCOL, PRETTYFLG, and CLISPIFYPRETTYFLG (see the Special Prettyprint Controls section of Chapter 26) are among the many variables controlling how functions are pretty printed.

List Structure Editor — The variable INITIALSLST (see the Time Stamps section of Chapter 16) is used when "time-stamps" are inserted in a function when it is edited. EDITCHARACTERS (see the Time Stamps section of Chapter 16) is used to set the read macros used in the teletype editor.

Idle Mode

The Medley environment runs on small single-user computers, usually located in users' offices. Often, users leave their computers up and running for days, which can cause several problems. First, the phosphor in the video display screen can be permanently marked if the same pattern is displayed for a long time (weeks). Second, if you go away, leaving a Medley system running, another person could possibly walk up and use the environment, taking advantage of any passwords that have been entered. To solve these problems, Medley implements the concept of "idle mode."

If no keyboard or mouse action has occurred for a specified time, Medley automatically enters idle mode. While idle mode is on, the display screen is blacked out, to protect the phosphor. Idle mode also runs a program to display some moving pattern on the black screen, so the screen does not appear to be broken. Usually, idle mode can be exited by pressing any key on the keyboard or mouse. However, you can optionally specify that idle mode should erase the current password cache when it is entered., and require the next user to supply a password to exit idle mode.

If either shift key is pressed while Medley is in idle mode, the current user name and the amount of time spent idling are displayed in the prompt window while the key is depressed.

Idle mode can also be entered by calling the function IDLE , or by selecting the Idle menu command from the background menu (see Chapter 28). The Idle menu command has subitems that allow you to interactively set the idle options (display program, erasing password, etc.) specified by the variable IDLE.PROFILE.

IDLE.PROFILE [Variable]

The value of this variable is a property list (see Chapter 3) which controls most aspects of idle mode. The following properties are recognized:

TIMEOUT Value is a number that determines how long (in minutes) Medley will wait before automatically entering idle mode. If NIL, idle mode will never be entered automatically. Default is 10 minutes.

FORGET If this is the symbol FIRST, your password will be erased when idle mode is entered. If non-NIL, your password will be erased when idle mode is exited. Initial value is T (erase password on exit).

If the password is erased on entry to idle mode (value FIRST), any programs left running when idle mode is entered will fail if they try doing anything requiring passwords (such as accessing file servers).

ALLOWED.LOGINS The value of this property can either be a list or a non-list. If the value is NIL or any other non-list, idle mode is exited without requesting login.

If the value is a list the members of the list should be interpreted as follows:

* If the value is a list containing * as it's element, login is required but anyone can exit idle mode. This will overwrite the previous user's user name and password each time idle mode is exited.

T Let the previous user (as determined by USERNAME) exit idle mode. If the username has not been set, this is equivalent to *

user name Let this specific user exit idle mode.

group name Let any member of this group (an NS clearinghouse group) exit idle mode.

AUTHENTICATE The value of this property determines the method used for logging in. The value can be one of the following:

T or NS Use the NS authentication protocol. This requires that you have an NS authentication server accessible on your net.

`GV`	Authenticate the login via the GrapeVine protocol.
`UNIX`	Use the unix login mechanism.
	Note: Unix is case sensitive. If you try to login but fail, you may have typed the password with the caps-lock on.
`LOGIN.TIMEOUT`	This is the number of seconds idle will wait for a login before resuming idle mode again.
`DISPLAYFN`	The value of this property, which should be a function name or lambda expression, is called to display a moving pattern on the screen while in idle mode. This function is called with one argument, a window covering the whole screen. The default is `IDLE.BOUNCING.BOX` (below).
	Any function used as a `DISPLAYFN` should call `BLOCK` (see Chapter 23) frequently, so other programs can run during idle mode.
`SAVEVM`	Value is a number that determines how long (in minutes) after idle mode is entered that `SAVEVM` will be called to save the virtual memory. If `NIL`, `SAVEVM` is never called automatically from idle mode. Default is 10 minutes.
`SUSPEND.PROCESS.NAMES`	Value is a list of names. For each name on this list, if a process by that name is found, it will be suspended upon entering idle mode and woken upon exiting idle mode.

IDLE.RESETVARS [Variable]

The value of this variable is a list of two-element lists: `((`VAR_1 EXP_1`)(`VAR_2 EXP_2`)...)`. On entering idle mode, each variable VAR_N is bound to the value of the corresponding expression EXP_N. When idle mode is exited, each variable VAR_N is reset to its original value.

IDLE.SUSPEND.PROCESS.NAMES [Variable]

Value is a list of names. For each name on this list, if a process by that name is found, it will be suspended upon entering idle mode and woken upon exiting idle mode.

IDLE.PROFILE [Variable]

The value of this variable determines the menu raised by selecting the Display subitem of the Idle background menu command. It should be in the format used for the `ITEMS` field of a menu (see Chapter 28), with the selection of an item returning the appropriate display function.

(**IDLE.BOUNCING.BOX** *WINDOW BOX WAIT*) [Variable]

This is the default display function used for idle mode. *BOX* is bounded about *WINDOW*, with bounces taking place every *WAIT* milliseconds. *BOX* can be a string, a bitmap, a window (whose image will be bounced about), or a list containing any number of these (which will be cycled through). *BOX* defaults to the value of the variable IDLE.BOUNCING.BOX, which is initially a bitmap of the Venue logo. *WAIT* defaults to 1000 (one second).

Saving Virtual Memory State

Medley storage allocation occurs within a virtual memory space that is usually much larger than the physical memory on the computer. The virtual memory is stored as a large file on the computer's hard disk, called the virtual memory file. Medley controls the swapping of pages between this file and the real memory, swapping in virtual memory pages as they are accessed, and swapping out pages that have been modified. At any moment, the total state of the Medley virtual memory is stored partially in the virtual memory file, and partially in the real physical memory.

Medley provides facilities for saving the total state of the virtual memory, either on the virtual memory file, or in a file on an arbitrary file device. The function LOGOUT is used to write all altered (dirty) pages from the real memory to the virtual memory file and stop Medley, so that Medley can be restarted from the state of the LOGOUT. SAVEVM updates the virtual memory file without stopping Medley, which puts the virtual memory file into a consistant state (temporarily), so it could be restarted if the system crashes. SYSOUT and MAKESYS are used to save a copy of the total virtual memory state on a file, which can be loaded into another machine to restore Medley's state. VMEM.PURE.STATE can be used to "freeze" the current state of the virtual memory, so that Medley will come up in that state if it is restarted.

(**LOGOUT** *FAST*) [Function]

Stops Medley, and returns control to the operating system. If Medley is restarted, it should come up in the same state as when the LOGOUT was called. LOGOUT will not affect the state of open files.

LOGOUT writes out all altered pages from real memory to the virtual memory file. If *FAST* is T, Medley is stopped without updating the virtual memory file. Note that after doing (LOGOUT T) it will not be possible to restart Medley from the point of the LOGOUT, and it may not be possible to restart it at all. Typing (LOGOUT T**)** is preferable to just booting the machine, because it also does other cleanup operations (closing network connections, etc.).

If *FAST* is the symbol ?, LOGOUT acts like *FLG* = T if the virtual memory file is consistant, otherwise it acts like *FLG* = NIL. This insures that the virtual memory image can be restarted as of *some* previous state, not necessarily as of the LOGOUT.

(**SAVEVM** —) [Function]

This function is similar to logging out and continuing, but faster. It takes about as long as a logout, which can be as brief as 10 seconds or so if you have already written out most of your dirty pages by virtue of being idle a while. After the SAVEVM, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the SAVEVM) should there be a system crash or other disaster.

If the system has been idle long enough (no keyboard or mouse activity), there are dirty pages to be written, and there are few enough dirty pages left to write that a SAVEVM would be quick, SAVEVM is automatically called. When SAVEVM is called automatically, the cursor is changed to a special cursor: , stored in the variable SAVINGCURSOR. You can control how often SAVEVM is automatically called by setting the following two global variables:

SAVEVMWAIT [Variable]
SAVEVMMAX [Variable]

The system will call SAVEVM after being idle for SAVEVMWAIT seconds (initially 300) if there are fewer than SAVEVMMAX pages dirty (initially 600). These values are fairly conservative. If you want to be extremely wary, you can set SAVEVMWAIT = 0 and SAVEVMMAX = 10000, in which case SAVEVM will be called the first chance available after the first dirty page has been written.

The function SYSOUT saves the current state of Medley's virtual memory on a file, known as a "sysout file", or simply a "sysout". The file package can be used to save particular function definitions and other arbitrary objects on files, but SYSOUT saves the *total* state of the system. This capability can be useful in many situations: for creating customized systems for other people to use, or to save a particular system state for debugging purposes. Note that a sysout file can be very large (thousands of pages), and can take a long time to create, so it is not to be done lightly. The file produced by SYSOUT can be loaded into Medley's virtual memory and restarted to restore the virtual memory to the exact state that it had when the sysout file was made. The exact method of loading a sysout depend on the implementation. For more information on loading sysout files, see the users guide for your computer.

(**SYSOUT** *FILE*) [Function]

Saves the current state of Medley's virtual memory on the file *FILE*, in a form that can be subsequently restarted. The current state of program execution is saved in the sysout file, so (PROGN (SYSOUT 'FOO) (PRINT 'HELLO)) will cause HELLO to be printed after the sysout file is restarted.

SYSOUT can take a very long time (ten or fifteen minutes), particularly when storing a file on a remote file server. To display some indication that something is happening, the cursor is changed to: . Also, as the sysout file is being written, the cursor is inverted line by line, to show that activity is taking place, and how much of the sysout has completed. For example, after the SYSOUT is about two-thirds done, the cursor would look like: . The SYSOUT cursor is stored in the variable SYSOUTCURSOR.

If *FILE* is non-NIL, the variable SYSOUTFILE is set to the body of *FILE*. If *FILE* is NIL, then the value of SYSOUTFILE instead. Therefore, (SYSOUT) will save the current state on the next higher version of a file with the same name as the previous SYSOUT. Also, if the extension for *FILE* is not specified, the value of SYSOUT.EXT is used. SYSOUT sets SYSOUTDATE (see the System Version Information section below) to (DATE), the time and date that the SYSOUT was performed.

If SYSOUT was not able to create the sysout file, because of disk or computer error, or because there was not enough space on the directory, SYSOUT returns NIL. Otherwise it returns the full file name of *FILE*.

Actually, SYSOUT "returns" twice; when the sysout file is first created, and when it is subsequently restarted. In the latter case, SYSOUT returns a list whose CAR is the full file name of *FILE*. For example, (if (LISTP (SYSOUT 'FOO)) then (PRINT 'HELLO)) will cause HELLO to be printed when the sysout file is restarted, but not when SYSOUT is initially performed.

Note: SYSOUT does not save the state of any open files. Use WHENCLOSE (see the Closing and Reopening Files section in Chapter 24) to associate certain operations with open files so that when a SYSOUT is started up, these files will be reopened, and file pointers repositioned.

SYSOUT evaluates the expressions on BEFORESYSOUTFORMS (see also AROUNDEXITFNS) before creating the sysout file. This variable initially includes expressions to:

1. Set the variables SYSOUTDATE and SYSOUTFILE as described above

2. Default the sysout file name *FILE* according to the values of the variables SYSOUTFILE and SYSOUT.EXT, as described above

3. Perform any necessary operations on open files as specified by calls to WHENCLOSE.

After a sysout file is restarted (but *not* when it is initially created), SYSOUT evaluates the expressions on AFTERSYSOUTFORMS (see also AROUNDEXITFNS). This initially includes expressions to:

1. Perform any necessary operations on previously-opened files as specified by calls to WHENCLOSE

2. Possibly print a message, as determined by the value of SYSOUTGAG (see below)

3. Call SETINITIALS to reset the initials used for time-stamping (see the Time Stamps section of Chapter 16).

AROUNDEXITFNS [Variable]

This variable provides a way to "advise" the system on cleanup and restoration activities to perform around LOGOUT, SYSOUT, MAKESYS and SAVEVM; It subsumes the functionality of BEFORESYSOUTFORMS, AFTERLOGOUTFORMS, etc. It's value is a list of functions (names) to call around every "exit" of the system. Each function is called with one argument, a symbol indicating which particular event is occuring. The symbols are:

BEFORLOGOUT The system is about to perform a LOGOUT.

BEFORESYSOUT
BEFOREMAKESYS
BEFORESAVEVM The system is about to perform a SYSOUT, MAKESYS or a SAVEVM.

AFTERLOGOUT
AFTERSYSOUT
AFTERMAKESYS
AFTERSAVEVM The system is starting up am image that was saved by performing a LOGOUT, SYSOUT, etc.

AFTERDOSYSOUT

AFTERDOMAKESYS
AFTERDOSAVEVM The system just made a copy of the virtual memory and saved it to disk. The image continues to run. These events only exist to allow you to negate the effects of saveing a copy of the virtual memory.

SYSOUTGAG [Variable]

The value of SYSOUTGAG determines what is printed when a sysout file is restarted. If the value of SYSOUTGAG is a list, the list is evaluated, and no additional message is printed. This allows you to print a message. If SYSOUTGAG is non-NIL and not a list, no message is printed. Finally, if SYSOUTGAG is NIL (its initial value), and the sysout file is being restarted by the same user that made the sysout originally, you are greeted by printing the value of HERALDSTRING (see below) followed by a greeting message. If the sysout file was made by a different user, a message is printed, warning that the currently-loaded user init file may be incorrect for the current user (see the Greeting and Initialization Files section above).

(**MAKESYS** *FILE NAME*) [Function]

Used to store a new Medley system on the "makesys file" *FILE*. Like SYSOUT, but before the file is made, the system is "initialized" by undoing the greet history, and clearing the display.

When the system is first started up, a "herald" is printed identifying the system, typically "Medley-XX DATE ...". If *NAME* is non-NIL, MAKESYS will use it instead of Medley-XX in the herald. MAKESYS sets HERALDSTRING to the herald string printed out.

MAKESYS also sets the variable MAKESYSDATE (see the next section below) to (DATE), i.e. the time and date the system was made.

Medley contains a routine that writes out dirty pages of the virtual memory during I/O wait, assuming that swapping has caused at least one dirty page to be written back into the virtual memory file (making it non-continuable). The frequency with which this routine runs is determined by:

BACKGROUNDPAGEFREQ [Variable]

This variable determines how often the routine that writes out dirty pages is run. The *higher* BACKGROUNDPAGEFREQ is set, the *greater* the time between running the dirty page writing routine. Initially it is set to 4. The lower BACKGROUNDPAGEFREQ is set, the less responsiveness you get at typein, so it may not be desirable to set it all the way down to 1.

(**VMEM.PURE.STATE** *X*) [NoSpread Function]

VMEM.PURE.STATE modifies the swapper's page replacement algorithm so that dirty pages are only written at the end of the virtual memory backing file. This "freezes" a given virtual memory state, so that Medley will come up in that state whenever it is restarted. This can be used to set up a "clean" environment on a pool machine, allowing each user to initialize the system simply by rebooting the computer.

The way to use VMEM.PURE.STATE is to set up the environment as you wish it to be "frozen," evaluate (VMEM.PURE.STATE T), and then call any function that saves the virtual memory state (LOGOUT, SAVEVM, SYSOUT, or MAKESYS). From that point on, whenever the system is restarted, it will return to the state as of the saving operation. Future LOGOUT, SAVEVM, etc. operations will not reset this state.

Note: When the system is running in "pure state" mode, it uses a significant amount of the virtual memory backing file to save the "frozen" memory image, so this will reduce the amount of virtual memory space available for use.

(VMEM.PURE.STATE) returns T if the system is running in "pure state" mode, NIL otherwise.

(**REALMEMORYSIZE**) [Function]

Returns the number of real memory pages in the computer.

(**VMEMSIZE**) [Function]

Returns the number of pages in use in the virtual memory. This is the roughly the same as the number of pages required to make a sysout file on the local disk (see SYSOUT, above).

LASTVMEMFILEPAGE [Variable]

Value is the total size of the virtual memory backing file. This variable is set when the system is started. You should *not* set it..

Note: When the virtual memory expands to the point where the virtual memory backing file is almost full, a break will occur with the warning message "Your virtual memory backing file is almost full. Save your work & reload asap." When this happens, it is strongly suggested that you save any important work and reload the system. If you continue working past this point, the system will start slowing down considerably, and it will eventually stop working.

System Version Information

Medley runs on a number of different machines, with many possible hardware configurations. There have been a number of different releases of the Medley software. These facts make it difficult to answer the important question "what software/hardware environment are you running?" when reporting bugs. The following functions allow the novice to collect this information.

(**PRINT-LISP-INFORMATION** *STREAM FILESTRING*) [NoSpread Function]

Prints out a summary of the software and hardware environment that Medley is running in, and a list of all loaded patch files:

```
Venue Medley version
Medley 2.0 sysout of 7-Oct-92 15:18:52 on mips,
Emulator created: 20-Nov-92, memory size: 0,
machine d022899 mo
based on Envos Medley version Medley 2.0 sysout of 7-Oct-
92 15:18:52,
Make-init dates: 7-Oct-92 11:07:17, 7-Oct-92 11:26:22
Patch files: NIL
```

STREAM is the stream used to print the summary. If not given, it defaults to T.

FILESTRING is a string used to determine what loaded files should be listed as "patch files." All file names on LOADEDFILELST (see the Noticing Files section of Chapter 17) that have *FILESTRING* as a substring as listed. If *FILESTRING* is not given, it defaults to the string "PATCH".

(**CL:LISP-IMPLEMENTATION-TYPE**) [Function]

Returns a string identifying the type of implementation that is running, e.g., "Medley".

(**CL:LISP-IMPLEMENTATION-VERSION**) [Function]

Returns a string identifying the version that is running. Currently gives the system name and date, e.g., "KOTO of 10-Sep-85 08:25:46".

This uses the variables MAKESYSNAME and MAKESYSDATE (below), so it will change if you use MAKESYS (see the Saving Virtual Memory State section above) to create a custom sysout file, or explicitly changes these variables.

(**CL:SOFTWARE-TYPE**) [Function]

Returns a string identifying the operating system that Interlisp is running under. Currently returns the string "Envos Medley".

(**CL:SOFTWARE-VERSION**) [Function]

Returns a string identifying the version of the operating system that Interlisp is running under. Currently, this returns the date that the Medley release was originally created, so it doesn't change over MAKESYS or SYSOUT.

(**CL:MACHINE-TYPE**) [Function]

Returns a string identifying the type of computer hardware that Medley is running on, i.e., "1108", "1132", "1186", "mips", etc.

(**CL:MACHINE-VERSION**) [Function]

Returns a string identifying the version of the computer hardware that Medley is running on. Currently returns the microcode version and real memory size.

(**CL:MACHINE-INSTANCE**) [Function]

Returns a string identifying the particular machine that Medley is running on. Currently returns the machine's NS address.

(**CL:SHORT-SITE-NAME**) [Function]

Returns a short string identifying the site where the machine is located. Currently returns (ETHERHOSTNAME) (if non-NIL) or the string "unknown".

(**CL:LONG-SITE-NAME**) [Function]

Returns a long string identifying the site where the machine is located. Currently returns the same as SHORT-SITE-NAME.

SYSOUTDATE [Variable]

Value is set by SYSOUT (see the Saving Virtual Memory State section above) to the date before generating a virtual memory image file.

MAKESYSDATE [Variable]

Value is set by MAKESYS (see the Saving Virtual Memory State section above) to the date before generating a virtual memory image file.

MAKESYSNAME [Variable]

Value is a symbol identifying the release name of the current Medley system, e.g., :MEDLEY.

(**SYSTEMTYPE**) [Function]

Allows programmers to write system-dependent code. SYSTEMTYPE returns a symbol corresponding to the implementation of Interlisp: D (for Medley), TOPS-20, TENEX, JERICO, or VAX.

In Medley, (SELECTQ (SYSTEMTYPE) ...) expressions are expanded at compile time so that this is an effective way to perform conditional compilation.

(**MACHINETYPE**) [Function]

Returns the type of machine that Medley is running on: either DORADO (for the Xerox 1132), DOLPHIN (for the Xerox 1100), DANDELION (for the Xerox 1108), DOVE (for the Xerox 1186), or MAIKO (for Unix, DOS, etc).

Date And Time Functions

(**DATE** *FORMAT*) [Function]

Returns the current date and time as a string with format "*DD-MM-YY HH:MMM:SS*", where *DD* is day, *MM* is month, *YY* year, *HH* hours, *MMM* minutes, *SS* seconds, e.g., "7-Jun-85 15:49:34".

If *FORMAT* is a date format as returned by DATEFORMAT (below), it is used to modify the format of the date string returned by DATE.

(**IDATE** *STR*) [Function]

STR is a date and time string. IDATE returns *STR* converted to a number such that if DATE1 is before (earlier than) DATE2, then (IDATE *DATE1*) < (IDATE *DATE2*). If STR is NIL, the current date and time is used.

Different Interlisp implementations can have different internal date formats. However, IDATE always has the essential property that (IDATE *X*) is less than (IDATE *Y*) if *X* is before *Y*, and (IDATE (GDATE *N*)) equals *N*. Programs which do arithmetic other than numerical comparisons between IDATE numbers may not work when moved from one implementation to another.

Generally, it is possible to increment an IDATE number by an integral number of days by computing a "1 day" constant, the difference between two convenient IDATEs, e.g. (IDIFFERENCE (IDATE "2-JAN-80 12:00") (IDATE "1-JAN-80 12:00")). This "1 day" constant can be evaluated at compile time.

IDATE is guaranteed to accept as input the dates that DATE will output. It will ignore the parenthesized day of the week (if any). IDATE also correctly handles time zone specifications for those time zones registered in the list TIME.ZONES (below).

(**GDATE** *DATE FORMAT* —) [Function]

Like DATE, except that *DATE* can be a number in internal date and time format as returned by IDATE. If *DATE* is NIL, the current time and date is used.

(**DATEFORMAT** KEY_1 ... KEY_N) [NLambda NoSpread Function]

DATEFORMAT returns a date format suitable as a parameter to DATE and GDATE. KEY_1 ... KEY_N are a set of keywords (unevaluated). Each keyword affects the format of the date independently (except for SLASHES and SPACES). If the date returned by (DATE) with the default formatting was 7-Jun-85 15:49:34, the keywords would affect the formatting as follows:

NO.DATE: Doesn't include the date information, e.g. "15:49:34".

NUMBER.OF.MONTH: Shows the month as a number instead of a name, e.g. "7-06-85 15:49:34".

YEAR.LONG: Prints the year using four digits, e.g. "7-Jun-1985 15:49:34".

SLASHES: Separates the day, month, and year fields with slashes, e.g. "7/Jun/85 15:49:34".

SPACES: Separates the day, month, and year fields with spaces, e.g. "7 Jun 85 15:49:34".

NO.LEADING.SPACES: By default, the day field will always be two characters long. If NO.LEADING.SPACES is specified, the day field will be one character for dates earlier than the 10th, e.g. "7-Jun-85 15:49:34" instead of "7-Jun-85 15:49:34".

NO.TIME: Doesn't include the time information, e.g. "7-Jun-85".

TIME.ZONE: Includes the time zone in the time specification, e.g. "7-Jun-85".

NO.SECONDS: Doesn't include the seconds, e.g. "7-Jun-85 15:49".

DAY.OF.WEEK: Includes the day of the week in the time specification, e.g. "7-Jun-85 15:49:34 PDT (Friday)".

DAY.SHORT — If DAY.OF.WEEK is specified to include the day of the week, the week day is shortened to the first three letters, e.g. "7-Jun-85 15:49:34 PDT (Fri)". Note that DAY.SHORT has no effect unless DAY.OF.WEEK is also specified.

(**CLOCK** *N* —) [Function]

If *N* = 0, CLOCK returns the current value of the time of day clock i.e., the number of milliseconds since last system start up.

If *N* = 1, returns the value of the time of day clock when you started up this Interlisp, i.e., difference between (CLOCK 0) and (CLOCK 1) is number of milliseconds (real time) since this Interlisp system was started.

If *N* = 2, returns the number of milliseconds of *compute* time since user started up this Interlisp (garbage collection time is subtracted off).

If *N* = 3, returns the number of milliseconds of compute time spent in garbage collections (all types).

(**SETTIME** *DT*) [Function]

Sets the internal time-of-day clock. If *DT* = NIL, SETTIME attempts to get the time from the communications net; if it fails, you are prompted for the time. If *DT* is a string in a form that IDATE recognizes, it is used to set the time.

The following variables are used to interpret times in different time zones. \TimeZoneComp, \BeginDST, and \EndDST are normally set automatically if your machine is connected to a network with a time server. For standalone machines, it may be necessary to set them by hand (or in your init file, see the first section of this chapter) if you are not in the Pacific time zone.

TIME.ZONES [Variable]

Value is an association list that associates time zone specifications (PDT, EST, GMT, etc.) with the number of hours west of Greenwich (negative if east). If the time zone specification is a single letter, it is appended to "DT" or "ST" depending on whether daylight saving time is in effect. Initially set to:

```
((8 . P) (7 . M) (6 . C) (5 . E) (0 . GMT))
```

This list is used by DATE and GDATE when generating a date with the TIME.ZONE format is specified, and by IDATE when parsing dates.

\TimeZoneComp [Variable]

This variable should be initialized to the number of hours west of Greenwich (negative if east). For the U.S. west coast it is 8. For the east coast it is 5.

\BeginDST [Variable]
\EndDST [Variable]

\BeginDST is the day of the year on or before which Daylight Savings Time takes effect (i.e., the Sunday on or immediately preceding this day); \EndDST is the day on or before which Daylight Savings Time ends. Days are numbered with 1 being January 1, and counting the days as for a leap year. In the USA where Daylight Savings Time is observed, `\BeginDST = 121` and `\EndDST = 305`. In a region where Daylight Savings Time is not observed at all, set `\BeginDST` to 367.

Timers and Duration Functions

Often one needs to loop over some code, stopping when a certain interval of time has passed. Some systems provide an "alarm clock" facility, which provides an asynchronous interrupt when a time interval runs out. This is not particularly feasible in the current Medley environment, so the following facilities are supplied for efficiently testing for the expiration of a time interval in a loop context.

Three functions are provided: `SETUPTIMER`, `SETUPTIMER.DATE`, and `TIMEREXPIRED?`. There are also several new i.s.oprs: `forDuration`, `during`, `untilDate`, `timerUnits`, `usingTimer`, and `resourceName` (reasonable variations on upper/lower case are permissible).

These functions use an object called a timer, which encodes a future clock time at which a signal is desired. A timer is constructed by the functions `SETUPTIMER` and `SETUPTIMER.DATE`, and is created with a basic clock "unit" selected from among `SECONDS`, `MILLISECONDS`, or `TICKS`. The first two timer units provide a machine/system independent interface, and the latter provides access to the "real", basic strobe unit of the machine's clock on which the program is running. The default unit is `MILLISECONDS`.

Currently, the `TICKS` unit depends on what machine Medley is running on. The Xerox 1132 has about 1680 ticks per millisecond; the Xerox 1108 has about 34.746 ticks per millisecond; the Xerox 1185 and 1186 have about 62.5 ticks per millisecond. The advantage of using `TICKS` rather than one of the uniform interfaces is primarily speed; e.g., it may take over 400 microseconds to read the milliseconds clock (a software facility that uses the real clock), whereas reading the real clock itself may take less than ten microseconds. The disadvantage of the `TICKS` unit is its short roll-over interval (about 20 minutes) compared to the `MILLISECONDS` roll-over interval (about two weeks), and also the dependency on particular machine parameters.

(**`SETUPTIMER`** *`INTERVAL OldTimer? timerUnits intervalUnits`*) [Function]

`SETUPTIMER` returns a timer that will "go off" (as tested by `TIMEREXPIRED?`) after a specified time-interval measured from the current clock time. `SETUPTIMER` has one required and three optional arguments:

`INTERVAL` must be a integer specifying how long an interval is desired. *`timerUnits`* specifies the units of measure for the interval (defaults to `MILLISECONDS`).

If *`OldTimer?`* is a timer, it will be reused and returned, rather than allocating a new timer. *`intervalUnits`* specifies the units in which the *`OldTimer?`* is expressed (defaults to the value of *`timerUnits`*.

(**`SETUPTIMER.DATE`** *`DTS OldTimer?`*) [Function]

`SETUPTIMER.DATE` returns a timer (using the `SECONDS` time unit) that will "go off" at a specified date and time. *`DTS`* is a Date/Time string such as `IDATE` accepts (see the above section). If *`OldTimer?`* is a timer, it will be reused and returned, rather than allocating a new timer.

SETUPTIMER.DATE operates by first subtracting (IDATE) from (IDATE *DTS*), so there may be some large integer creation involved, even if *OLDTIMER?* is given.

(**TIMEREXPIRED?** *TIMER* *ClockValue.or.timerUnits*) [Function]

If *TIMER* is a timer, and *ClockValue.or.timerUnits* is the time-unit of *TIMER*, TIMEREXPIRED? returns true if *TIMER* has “gone off”.

ClockValue.or.timerUnits can also be a timer, in which case TIMEREXPIRED? compares the two timers (which must be in the same timer units). If X and Y are timers, then (TIMEREXPIRED? X Y) is true if X is set for an *earlier* time than Y.

There are a number of i.s.oprs that make it easier to use timers in iterative statements (see the Interative Statement section of Chapter 9). These i.s.oprs are given below in the “canonical” form, with the second “word” capitalized, but the all-caps and all-lower-case versions are also acceptable.

forDuration *INTERVAL* [I.S. Operator]
during *INTERVAL* [I.S. Operator]

INTERVAL is an integer specifying an interval of time during which the iterative statement will loop.

timerUnits *UNITS* [I.S. Operator]

UNITS specifies the time units of the *INTERVAL* specified in forDuration.

untilDate *DTS* [I.S. Operator]

DTS is a Date/Time string (such as IDATE accepts) specifying when the iterative statement should stop looping.

usingTimer *TIMER* [I.S. Operator]

If usingTimer is given, *TIMER* is reused as the timer for forDuration or untilDate, rather than creating a new timer. This can reduce allocation if one of these i.s.oprs is used within another loop.

resourceName *RESOURCE* [I.S. Operator]

RESOURCE specifies a resource name to be used as the timer storage (see the File Package Types section of Chapter 17). If *RESOURCE* = T, it will be converted to an internal name.

Some examples:

```
(during 6MONTHS timerUnits 'SECONDS
  until (TENANT-VACATED? HouseHolder)
  do (DISMISS <for-about-a-day>)
     (HARRASS HouseHolder)
  finally (if (NOT (TENANT-VACATED? HouseHolder))
           then (EVICT-TENANT HouseHolder)))
```

This example shows that how is is possible to have two termination condition: when the time interval of 6MONTHS has elapsed, or when the predicate (TENANT-VACATED? HouseHolder) becomes true. Note that the “finally” clause is executed regardless of which termination condition caused it.

Also note that since the millisecond clock will "roll over" about every two weeks, "6MONTHS" wouldn't be an appropriate interval if the timer units were the default case, namely MILLISECONDS.

```
(do (forDuration (CONSTANT (ITIMES 10 24 60 60 1000))
      do (CARRY.ON.AS.USUAL)
      finally (PROMPTPRINT "Have you had your 10-day check-up?")))
```

This infinite loop breaks out with a warning message every 10 days. One could question whether the millisecond clock, which is used by default, is appropriate for this loop, since it rolls-over about every two weeks.

```
(SETQ \RandomTimer (SETUPTIMER 0))
 (untilDate "31-DEC-83 23:59:59"  usingTimer \RandomTimer
  when (WINNING?) do (RETURN)
  finally (ERROR "You've been losing this whole year!"))
```

Here is a usage of an explicit date for the time interval; also, some stsorage has been squirreled away (as the value of \RandomTimer) for use by the call to SETUPTIMER in this loop.

```
(forDuration SOMEINTERVAL
  resourceName \INNERLOOPBOX
  timerunits 'TICKS
  do (CRITICAL.INNER.LOOP))
```

For this loop, you don't want any CONSing to take place, so \INNERLOOPBOX is defined as a resource which "caches" a timer cell (if it isn't already so defined), and wraps the entire statement in a WITH-RESOURCES call. Furthermore, a time unit of TICKS is specified, for lower overhead in this critical inner loop. In fact specifying a resourceName of T is the same as specifying it to be \ForDurationOfBox; this is just a simpler way to specify that a resource is wanted, without having to think up a name.

Resources

Medley is based on the use of a storage-management system which allocates memory space for new data objects, and automatically reclaims the space when no longer in use. More generally, Medley manages shared "resources", such as files, semaphors for processes, etc. The protocols for allocating and freeing such resources resemble those of ordinary storage management.

Sometimes you need to explicitly manage the allocation of resources. You may want the efficiency of explicit reclamation of certain temporary data; or it may be expensive to initialize a complex data object; or there may be an application that must not allocate new cells during some critical section of code.

The file manager type RESOURCES is available to help with the definition and usage of such classes of data; the definition of a RESOURCE specifies prototype code to do the basic management operations. The file manager command RESOURCES is used to save such definitions on files, and INITRESOURCES (see the Miscellaneous File Manager Commands section of Chapter 17) causes the initialization code to be output.

The basic needs of resource management are:

1. Obtaining a data item from the Lisp memory management system and configuring it to be a totally new instance of the resource in question
2. Freeing up an instance which is no longer needed

3. Getting an instance of the resource for temporary usage (whether “fresh” or a formerly freed-up instance)
4. Setting up any prerequisite global data structures and variables

A resources definition consists of four “methods”: INIT, NEW, GET, and FREE; each “method” is a form that will specialize the definition for four corresponding user-level macros INITRESOURCE, NEWRESOURCE, GETRESOURCE, and FREERESOURCE. PUTDEF is used to make a resources definition, and the four components are specified in a proplist:

```
(PUTDEF
    'RESOURCENAME
    'RESOURCES
    '(NEW  NEW-INSTANCE-GENERATION-CODE
      FREE FREEING-UP-CODE
      GET  GET-INSTANCE-CODE
      INIT INITIALIZATION-CODE))
```

Each of the *xxx-CODE* forms is a form that will appear as if it were the body of a substitution macro definition for the corresponding macro (see the discussion on the macros below).

A Simple Example

Suppose one has several pieces of code which use a 256-character string as a scratch string. One could simply generate a new string each time, but that would be inefficient if done repeatedly. If you can guarantee that there are no re-entrant uses of the scratch string, then it could simply be stored in a global variable. However, if the code might be re-entrant on occasion, the program has to take precautions that two programs do not use the same scratch string at the same time. (This consideration becomes very important in a multi-process environment. It is hard to guarantee that two processes won’t be running the same code at the same time, without using elaborate locks.) A typical tactic would be to store the scratch string in a global variable, and set the variable to NIL whenever the string is in use (so that re-entrant usages would know to get a “new” instance). For example, assuming the global variable TEMPSTRINGBUFFER is initialized to NIL:

```
[DEFINEQ (WITHSTRING NIL
   (PROG ((BUF (OR (PROG1 TEMPSTRINGBUFFER
                (SETQ TEMPSTRINGBUFFER NIL))
          (ALLOCSTRING 256))))
```

... use the scratch string in the variable BUF ...

```
      (SETQ TEMPSTRINGBUFFER BUF)
       (RETURN]
```

Here, the basic elements of a “resource” usage may be seen:

1. A call (ALLOCSTRING 256) allocates fresh instances of “buffer”
2. After usage is completed the instance is returned to the “free” state, by putting it back in the global variable TEMPSTRINGBUFFER where a subsequent call will find it
3. The prog-binding of BUF will get an existing instance of a string buffer if there is one -- otherwise it will get a new instance which will later be available for reuse
4. Some initialization is performed before usage of the resource (in this case, it is the setting of the global variable TEMPSTRINGBUFFER).

Given the following resources definition:

```
(PUTDEF
    'STRINGBUFFER
    'RESOURCES
    '(NEW  (ALLOCSTRING 256)
      FREE (SETQ TEMPSTRINGBUFFER (PROG1 . ARGS))
      GET  (OR (PROG1 TEMPSTRINGBUFFER
                    (SETQ TEMPSTRINGBUFFER NIL))
               (NEWRESOURCE TEMPSTRINGBUFFER)))
      INIT (SETQ TEMPSTRINGBUFFER NIL)))
```

we could then redo the example above as

```
(DEFINEQ (WITHSTRING NIL
    (PROG ((BUF (GETRESOURCE STRINGBUFFER)))
```

... use the string in the variable BUF ...

```
      (FREERESOURCE STRINGBUFFER BUF)
       (RETURN]
```

The advantage of doing the coding this way is that the resource management part of WITHSTRING is fully contained in the expansions of GETRESOURCE and FREERESOURCE, and thus there is no confusion between what is WITHSTRING code and what is resource management code. This particuar advantage will be multiplied if there are other functions which need a “temporary” string buffer; and of course, the resultant modularity makes it much easier to contemplate minor variations on, as well as multiple clients of, the STRINGBUFFER resource.

In fact, the scenario just shown above in the WITHSTRING example is so commonly useful that an abbreviation has been added; if a resources definition is made with *only* a NEW method, then appropriate FREE, GET, and INIT methods will be inferred, along with a coordinated globalvar, to be parallel to the above definition. So the above definition could be more simply written

```
(PUTDEF 'STRINGBUFFER
         'RESOURCES
         '(NEW  (ALLOCSTRING 256)))
```

and everything would work the same.

The macro WITH-RESOURCES simplifies the common scoping case, where at the beginning of some piece of code, there are one or more GETRESOURCE calls the results of which are each bound to a lambda variable; and at the ending of that code a FREERESOURCE call is done on each instance. Since the resources are locally bound to variables with the same name as the resource itself, the definition for WITHSTRING then simplifies to

```
(DEFINEQ (WITHSTRING NIL
    (WITH-RESOURCES (STRINGBUFFER)
```

... use the string in the variable STRINGBUFFER ...]

Trade-offs in More Complicated Cases

This simple example presumes that the various functions which use the resource are generally not re-entrant. While an occasional re-entrant use will be handled correctly (another example of the resource will simply be created), if this were to happen too often, then many of the resource requests will create and throw away new objects, which defeats one of the major purposes of using resources. A slightly more complex GET and FREE method can be of much benefit in maintaining a pool of available resources; if the resource were defined to maintain a list of "free" instances, then the GET method could simply take one off the list and the FREE method could just push it back onto the list. In this simple example, the SETQ in the FREE method defined above would just become a "push", and the first clause of the GET method would just be (pop TEMPSTRINGBUFFER)

A word of caution: if the datatype of the resource is something very small that Medley is "good" at allocating and reclaiming, then explicit user storage management will probably not do much better than the combination of cons / createcell and the garbage collector. This would especially be so if more complicated GET and FREE methods were to be used, since their overhead would be closer to that of the built-in system facilities. Finally, it must be considered whether retaining multiple instances of the resource is a net gain; if the re-entrant case is truly rare, it may be more worthwhile to retain at most one instance, and simply let the instances created by the rarely-used case be reclaimed in the normal course of garbage collection.

Macros for Accessing Resources

Four user-level macros are defined for accessing resources:

(**NEWRESOURCE** *RESOURCENAME* . *ARGS*) [Macro]
(**FREERESOURCE** *RESOURCENAME* . *ARGS*) [Macro]
(**GETRESOURCE** *RESOURCENAME* . *ARGS*) [Macro]
(**INITRESOURCE** *RESOURCENAME* . *ARGS*) [Macro]

Each of these macros behave as if they were defined as a substitution macro of the form

((*RESOURCENAME* . *ARGS*) *MACROBODY*)

where the expression *MACROBODY* is selected by using the "code" supplied by the corresponding method from the *RESOURCENAME* definition.

Note that it is possible to pass "arguments" to your resource allocation macros. For example, if the GET method for the resource FOO is (GETFOO . ARGS), then (GETRESOURCE FOO X Y) is transformed into (GETFOO X Y). This form was used in the FREE method of the STRINGBUFFER resource described above, to pass the old STRINGBUFFER object to be freed.

(**WITH-RESOURCES** (*RESOURCE*$_1$ *RESOURCE*$_2$ ...) *FORM*$_1$ *FORM*$_2$ ...) [Macro]

The WITH-RESOURCES macro binds lambda variables of the same name as the resources (for each of the resources *RESOURCE*$_1$, *RESOURCE*$_2$, etc.) to the result of the GETRESOURCE macro; then executes the forms *FORM*$_1$, *FORM*$_2$, etc., does a FREERESOURCE on each instance, and returns the value of the last form (evaluated and saved before the FREERESOURCEs).

Note: (WITH-RESOURCES *RESOURCE* ...) is interpreted the same as (WITH-RESOURCES (*RESOURCE*) ...). Also, the singular name WITH-RESOURCE is accepted as a synonym for WITH-RESOURCES.

Saving Resources in a File

Resources definitions may be saved on files using the RESOURCES file package command (see the Miscellaneous File Package Commands section of Chapter 17). Typically, one only needs the full definition available when compiling or interpreting the code, so it is appropriate to put the file package command in a (DECLARE: EVAL@COMPILE DONTCOPY ...) declaration, just as one might do for a RECORDS declaration. But just as certain record declarations need *some* initialization in the run-time environment, so do most resources. This initialization is specified by the resource's INIT method, which is executed automatically when the resource is defined by the PUTDEF output by the RESOURCES command. However, if the RESOURCES command is in a DONTCOPY expression and thus is not included in the compiled file, then it is necessary to include a separate INITRESOURCES command (see the Miscellaneous File Manager Commands section of Chapter 17) in the filecoms to insure that the resource is properly initialized.

[This page intentionally left blank]

13. MEDLEY EXECUTIVES

In most Common Lisp implementations, there is a “top-level read-eval-print loop,” which reads an expression, evaluates it, and prints the results. In Medley, the Exec acts as the top-level loop, but does much more.

The Exec traps all THROWS, and recovers gracefully. It prints all values resulting from evaluation, on separate lines. (When zero values are returned, nothing is printed).

The Exec keeps track of your previous inputs, in the history list. Each entry you type creates a history event, which sotres the input and its values.

It’s easy to use the results of earlier events, redo and event, or recall an earlier input, edit it, and run it. This makes it much easier to get your work done.

Multiple Execs and the Exec’s Type

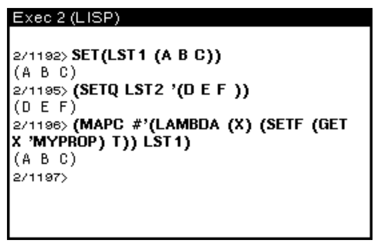
Sometimes you need more than one Exec open at a time. It’s easy to open as many as you need by using the right button background menu and selecting the kind of Exec you need. The Execs are differentiated from one another by their "names" in their title bars and by their prompts. For example, the second Exec you open may have a prompt like 2/50> if it’s the second Common Lisp Exec you’ve opened. Events in each Exec are placed on the global history list with their Exec number so the system can tell them apart.

Several variables are very important to an Exec since they control the format of reading and printing. Together these variables describe a type of exec, or its mode. Some standard bindings for the variables have been named to make mode setting easy. The names provide you with an Exec of the Common Lisp (LISP), Interlisp or Old Interlisp (IL), or Medley (XCL) type. An Exec’s type is displayed in the title bar of its window:

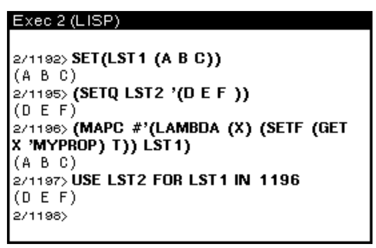
A Brief Example of Exec Interactions

The following dialogue contains examples and gives the flavor of the use of an Exec. The commands are described in greater detail in the following sections. For now, be sure to type these examples to an Exec whose *PACKAGE* is set to the XCL-USER package. The Exec that Medley starts up with is set to the XCL-USER package. Each prompt consists of an Exec number, an event number and a prompt character (“>” for Common Lisp and “←” for Interlisp).

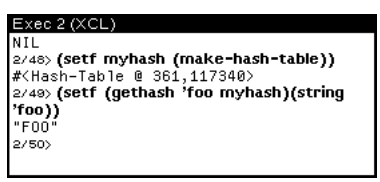
You have instructed the Exec to UNDO the previous event.



The Exec accepts input both in `APPLY` format (the `SET`) and `EVAL` format (the `SETQ`). In event 1196, you added a property `MYPROP` to the symbols `A`, `B`, and `C`.

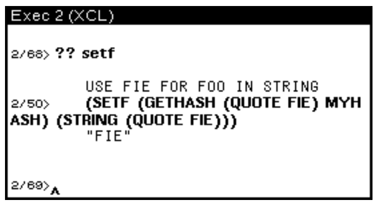


You told the Exec to go back to event 1196, substitute `LST2` for `LST1`, and then re-execute the expression.



If `STRING` were computationally expensive (it isn't), you might be caching its value for later use.

You now decide you would like to redo the `SETF` with a different value. You can specify the event using any symbol in the expression.



Here you ask the Exec (using the ?? command) what it has on its history list for the last input. Since the event corresponds to a command, the Exec displays both the original command and the generated input.

You'll usually deal with the Exec at top level or in the debugger, where you type in expressions for evaluation, and see the values printed out. An Exec acts much like a standard Lisp top-level loop, but before it evaluates an input, it first adds it to the history list. If the operation is aborted or causes an error, the input is still available for you to modify or re-execute.

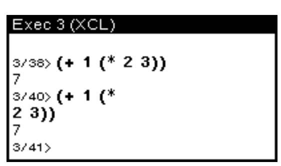
After updating the history list, the Exec executes the computation (i.e., evaluates the form or applies the function to its arguments), saves the value in the history-list entry for that input, and prints the result. Finally the Exec displays a prompt to show it's again ready for input.

Input Formats

The Exec accepts three forms of input: an expression to be evaluated (EVAL-format), a function-name and arguments to apply it to (APPLY-format), and Exec commands, as follows:

EVAL-format input — If you type a single expression, either followed by a carriage-return, or, in the case of a list, terminated with balanced parenthesis, the expression is evaluated and the value is returned. For example, if the value of FOO is the list (A B C):

Similarly, if you type a Lisp expression, beginning with a left parenthesis and terminated by a matching right parenthesis, the form is simply passed to EVAL for evaluation. Notice that it is not necessary to type a carriage return at the end of such a form; the reader will supply one automatically. If a carriage-return is typed before the final matching right parenthesis or bracket, it is treated the same as a space, and input continues. The following examples are interpreted identically:



APPLY-format input — Often, you call functions with constant argument values, which would have to be quoted if you typed them in EVAL-format. For convenience, if you type a symbol immediately followed by a list, the symbol is APPLYed to the elements within the list, unevaluated. The input is terminated by the matching right parenthesis. For example, typing LOAD(FOO) is equivalent to typing (LOAD 'FOO), and GET(X

COLOR) is equivalent to (GET 'X 'COLOR). As a simple special case, a single right parenthesis is treated as a balanced set of parentheses, e.g. UNBREAK) is equivalent to UNBREAK()

The reader will only supply the "carriage return" automatically if no space appears between the initial symbol and the list that follows; if there is a space after the initial symbol on the line and the list that follows, the input is not terminated until you type a carriage return.

The Exec will not consider unparenthesized input with more than one argument to be in apply format, e.g.:

LIST(1) is apply format (executes after closing parenthesis is typed)

LIST (1) is apply format (second argument is a list, no trailing arguments given)

LIST '(1) 2 3 is NOT apply format, arguments are evaluated

LIST 1 2 3 is NOT apply format, arguments are evaluated

LIST 1not legal input: second argument is not a list

Note that APPLY-format input cannot be used for macros or special forms.

Exec commands The Exec recognizes a number of commands, which usually refer to past events on the history list. These commands are treated specially; for example, they may not be put on the history list. The format of a command is always a line beginning with the command name. (The Exec looks up the command name independent of package.) The remainder of the line, if any, is treated as "arguments" to the command. For example,

```
128> UNDO
mapc undone
```

```
129> UNDO (FOO --)
foo undone
```

are both valid command inputs.

Event Specification

Exec commands, like UNDO, frequently refer to previous events in the session's history. All Exec commands use the same conventions and syntax for indicating which event(s) the command refers to. This section shows you the syntax used to specify previous events.

An event address identifies one event on the history list. For example, the event address 42 refers to the event with event number 42, and -2 refers to two events back in the current Exec. Usually, an event address will contain only one or two commands.

Event addresses can be concatenated. For example, if FOO refers to event N, FOO FIE will refer to the first event before event N which contains FIE.

The symbols used in event addresses (such as AND, F, etc.) are compared with STRING-EQUAL, so that it does not matter what the current package is when you type an event address symbol to an Exec.

Specifications used below of the form *EventAddress* refer to event addresses, as described above. Since an event address may contain multiple words, the event address is parsed by searching for the words which delimit it. For example, in $EventAddress_1$ AND $EventAddress_2$, the notation $EventAddress_1$ corresponds to all words up to the AND in the event specification, and $EventAddress_2$ to all words after the AND in the event specification.

Event addresses are interpreted as follows:

N (an integer) If N is positive, it refers to the event with event number N (no matter which Exec the event occurred in.) If N is negative, it always refers to the event -N events backwards, counting only events belonging to the current Exec.

F Specifies that the next object in the event address is to be searched for, regardless of what it is. For example, F -2 looks for an event containing -2.

FROM *EventAddress*

All events since *EventAddress*, inclusive. For example, if there is a single Exec and the current event is number 53, then FROM 49 specifies events 49, 50, 51, and 52. FROM includes events from *all* Execs.

ALL *EventAddress*

Specifies all events satisfying *EventAddress*. For example, ALL LOAD, ALL SUCHTHAT FOO-P.

empty If nothing is specified, it is the same as specifying -1, i.e., the last event in the current Exec.

$EventSpec_1$ AND $EventSpec_2$ AND . . . AND $EventSpec_N$

Each of the is an event specification. The lists of events are concatenated. For example, REDO ALL MAPC AND ALL STRING AND 32 redoes all events containing MAPC, all containing STRING, and also event 32. Duplicate events are removed.

Exec Commands

You enter an Exec commands by typing the name of the command at the prompt. The name of an Exec command is not a symbol and therefore is not sensitive to the setting of the current package (the value of *PACKAGE*).

EventSpec is used to denote an event specification which in most cases will be either a specific event address (e.g., 42) or a relative one (e.g., -3). Unless specified otherwise, omitting *EventSpec* is the same as specifying *EventSpec* = -1. For example, REDO and REDO -1 are the same.

REDO *EventSpec* [Exec command]

Redoes the event or events specified by *EventSpec*. For example, REDO 123 redoes the event numbered 123.

RETRY *EventSpec* [Exec command]

Like REDO but sets the debugger parameters so that any errors that occur while executing *EventSpec* will cause breaks.

USE *NEW* [**FOR** *OLD*] [**IN** *EventSpec*] [Exec command]

Substitutes *NEW* for *OLD* in the events specified by *EventSpec*, and redoes the result. *NEW* and *OLD* can include lists or symbols, etc.

For example, `USE SIN (- X) FOR COS X IN -2 AND -1` will substitute `SIN` for every occurrence of `COS` in the previous two events, and substitute `(- X)` for every occurrence of `X`, and reexecute them. (The substitutions do not change the previous information saved about these events on the history list.)

If `IN` *EventSpec* is omitted, the first member of *OLD* is used to search for the appropriate event. For example, `USE DEFAULTFONT FOR DEFLATFONT` is equivalent to `USE DEFAULTFONT FOR DEFLATFONT IN F DEFLATFONT`. The `F` is inserted to handle the case where the first member of *OLD* could be interpreted as an event address command.

If *OLD* is omitted, substitution is for the "operator" in that command. For example `FBOUNDP(FF)` followed by `USE CALLS` is equivalent to `USE CALLS FOR FBOUNDP IN -1`.

If *OLD* is not found, `USE` will print a question mark, several spaces and the pattern that was not found. For example, if you specified `USE Y FOR X IN 104` and `X` was not found, "`X ?`" is printed to the Exec.

You can also specify more than one substitution simultaneously as follows:

USE *NEW*$_1$ **FOR** *OLD*$_1$ **AND** ... **AND** *NEW*$_N$ **FOR** *OLD*$_N$ [**IN** *EventSpec*] [Exec command]

[The `USE` command is parsed by a small finite state parser to distinguish the expressions and arguments. For example, `USE FOR FOR AND AND AND FOR FOR` will be parsed correctly.]

Every `USE` command involves three pieces of information: the expressions to be substituted, the arguments to be substituted for, and an event specification that defines the input expression in which the substitution takes place. If the `USE` command has the same number of expressions as arguments, the substitution procedure is straightforward. For example, `USE X Y FOR U V` means substitute `X` for `U` and `Y` for `V`, and is equivalent to `USE X FOR U AND Y FOR V`.

However, the `USE` command also permits distributive substitutions for substituting several expressions for the same argument. For example, `USE A B C FOR X` means first substitute `A` for `X` then substitute `B` for `X` (in a new copy of the expression), then substitute `C` for `X`. The effect is the same as three separate `USE` commands.

Similarly, `USE A B C FOR D AND X Y Z FOR W` is equivalent to `USE A FOR D AND X FOR W`, followed by `USE B FOR D AND Y FOR W`, followed by `USE C FOR D AND Z FOR W`. `USE A B C FOR D AND X FOR Y` also corresponds to three substitutions, the first with `A` for `D` and `X` for `Y`, the second with `B` for `D`, and `X` for `Y`, and the third with `C` for D, and again X for Y. However, USE A B C FOR D AND X Y FOR Z is ambiguous and will cause an error.

Essentially, the USE command operates by proceeding from left to right handling each AND separately. Whenever the number of expressions exceeds the available expressions, multiple USE expressions are generated. Thus USE A B C D FOR E F means substitute A for E at the same time substituting B for F, then in another copy of the indicated expression, substitute C for E and D for F. This is also equivalent to USE A C FOR E AND B D FOR F.

The USE command correctly handles the situation where one of the old expressions is the same as one of the new ones, USE X Y FOR Y X, or USE X FOR Y AND Y FOR X.

? *NAME* [Exec command]

If *NAME* is not provided describes all available Exec commands by printing the name, argument list, and description of each. With *NAME*, only that command is described.

?? *EventSpec* [Exec command]

Prints the most recent event matching the given *EventSpec*. Without *EventSpec*, lists all entries on the history list from all execs, not necessarily in the order in which they occured (since the list is in allocation order). If you haven't completed typing a command it will be listed as "<in progress>" .

Note: Event nubmers are allocated at the time the prompt is printed, except in the Old Interlisp exec where they are assigned at the end of type-in. This means that if activity occurs in another exec, the number printed next to the command is not necessarily the number associated with the event.

CONN *DIRECTORY* [Exec command]

Changes default pathname to *DIRECTORY*.

DA [Exec command]

Returns current date and time.

DIR *PATHNAME KEYWORDS* [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: *AUTHOR*, *AU*, *CREATIONDATE*, *DA*, etc.

DO-EVENTS *INPUTS ENV* [Exec command]

DO-EVENTS is intended as a way of putting together several different events, which can include commands. It executes the multiple *INPUTS* as a single event. The values returned by the *DO-EVENTS* event are the concatenation of the values of the inputs. An input is not an EventSpec, but a call to a function or command. If *ENV* is provided it is a lexical environment in which all evaluations (functions and commands) will take place. Event specification in the *INPUTS* should be explicit, not relative, since referring to the last event will reinvoke the executing *DO-EVENTS* command.

FIX *EventSpec* [Exec command]

Edits the specified event prior to re-executing it. If the number of characters in the fixed line is less than the variable TTYINFIXLIMIT then it will be edited using TTYIN, otherwise the Lisp editor is called via EDITE.

FORGET *EventSpec* [Exec command]

Erases UNDO information for the specified events.

NAME *COMMAND-NAME ARGUMENTS EVENT-SPEC* [Exec command]

Defines a new command, *COMMAND-NAME*, and its *ARGUMENTS*, containing the events in *EVENT-SPEC*.

NDIR *PATHNAME KEYWORDS* [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory in abbreviated format. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: AUTHOR, AU, CREATIONDATE, DA, etc.

PL *SYMBOL* [Exec command]

Prints the property list of *SYMBOL* in an easy to read format.

REMEMBER *&REST EVENT-SPEC* [Exec command]

Tells File Manager to remember type-in from specified event(s), *EVENT-SPEC*, as expressions to save.

SHH *LINE* [Exec command]

Executes *LINE* without history list processing.

UNDO *EventSpec* [Exec command]

Undoes the side effects of the specified event (see below under "Undoing").

PP *NAME TYPES* [Exec command]

Shows (prettyprinted) the definitions for *NAME* specified by *TYPES*.

SEE *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, hiding comments.

SEE* *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, showing comments.

TIME *FORM &KEY REPEAT &ENVIRONMENT ENV* [Exec command]

Times the evaluation of *FORM* in the lexical environment *ENV*, repeating *REPEAT* number of times. Information is displayed in the Exec window.

TY *FILES* [Exec command]

Exactly like the *TYPE* Exec command.

TYPE *FILES* [Exec command]

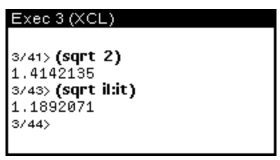
Prints the contents of *FILES* in the Exec window, hiding comments.

Variables

A number of variables are provided for convenience in the Exec.

IL:IT [Variable]

Whenever an event is completed, the global value of the variable IT is reset to the event's value. For example,



Following a ?? command, IL:IT is set to the value of the last event printed. The inspector has an option for setting the variable IL:IT to the current selection or inspected object, as well. The variable IL:IT is global, and is shared among all Execs. IL:IT is a convenient mechanism for passing values from one process to another.

Note: IT is in the Interlisp package and these examples are intended for an Exec whose *PACKAGE* is set to XCL-USER. Thus, IT must be package qualified (the IL:).

The following variables are maintained independently by each Exec. (When a new Exec is started, the initial values are NIL, or, for a nested Exec, the value for the "parent" Exec. However, events executed under a nested Exec will not affect the parent values.)

CL:- [Variable]
CL:+ [Variable]
CL:++ [Variable]
CL:+++ [Variable]

While a form is being evaluated by the Exec, the variable CL:- is bound to the form, CL:+ is bound to the previous form, CL:++ the one before, etc. If the input is in apply-format rather than eval-format, the value of the respective variable is just the function name.

CL:* [Variable]
CL:** [Variable]
CL:*** [Variable]

While a form is being evaluated by the Exec, the variable CL:* is bound to the (first) value returned by the last event, CL:** to the event before that, etc. The variable CL:* differs from IT in that IT is global while each separate Exec maintains its own copy of CL:*, CL:** and CL:***. In addition, the history commands change IT, but only inputs that are retained on the history list can change CL:*.

CL:/ [Variable]
CL:// [Variable]
CL:/// [Variable]

While a form is being evaluated by an Exec, the variable CL:/ is bound to a list of the results of the last event in that Exec, CL:// to the values of the event before that, etc.

Fonts in the Exec

The Exec can use different fonts for displaying the prompt, user's input, intermediate printout, and the values returned by evaluation. The following variables control the Exec's font use:

PROMPTFONT [Variable]

Font used for printing the event prompt.

INPUTFONT [Variable]

Font used for echoing your type-in.

PRINTOUTFONT [Variable]

Font used for any intermediate printing caused by execution of a command or evaluation of a form. Initially the same as DEFAULTFONT.

VALUEFONT [Variable]

Font used to print the values returned by evaluation of a form. Initially the same as DEFAULTFONT.

Modifying an Exec

(**CHANGESLICE** *N HISTORY —*) [Function]

Changes the maximum number of events saved on the history list *HISTORY* to *N*. If NIL, *HISTORY* defaults to the top level history LISPXHISTORY.

The effect of *increasing* the time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* the time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, CHANGESLICE is undoable, so that these events are (temporarily) recoverable. Therefore, if you want to recover the storage associated with these events without waiting *N* more events until the CHANGESLICE event drops off the history list, you must perform a FORGET command.

Defining New Commands

You can define new Exec commands using the XCL:DEFCOMMAND macro.

(**XCL:DEFCOMMAND** *NAME ARGUMENT-LIST &REST BODY*) [Macro]

XCL:DEFCOMMAND is like XCL:DEFMACRO, but defines new Exec commands. The *ARGUMENT-LIST* can have keywords, and use all of the features of macro argument lists. When *NAME* is subsequently typed to the Exec, the rest of the line is processed like the arguments to a macro, and the *BODY* is executed. XCL:DEFCOMMAND is a definer; the

File Manager will remember typed-in definitions and allow them to be saved, edited with EDITDEF, etc.

There are three kinds of commands that can be defined, :EVAL, :QUIET, and :INPUT. Commands can also be marked as only for the debugger, in which case they are labelled as :DEBUGGER. The command type is noted by supplying a list for the *NAME* argument to XCL:DEFCOMMAND, where the first element of the list is the command name, and the other elements are keyword(s) for the command type and, optionally :DEBUGGER.

The documentation string in user defined Exec commands is automatically added to the documentation descriptions by the CL:DOCUMENTATION function under the COMMANDS type and can be shown using the ? Exec command.

:EVAL This is the default. The body of the command just gets executed, and its value is the value of the event. For example (in an XCL Exec),

```
Exec 3 (XCL)

3/47> (DEFCOMMAND (LS :EVAL)
       (&OPTIONAL (NAMESTRING
*DEFAULT-PATHNAME-DEFAULTS*)
        &REST DIRECTORY-KEYWORDS)
       (MAPC #'(LAMBDA (PATHNAME)
          (FORMAT T "~&~A" (NAMESTRING PATHNAME)))
       (APPLY #'DIRECTORY NAMESTRING
DIRECTORY-KEYWORDS))
  (VALUES))
LS
3/48>
```

would define the LS command to print out all file names that match the input NAMESTRING. The (VALUES) means that no value will be printed by the event, only the intermediate output from the FORMAT.

:QUIET These commands are evaluated, but neither your input nor the results of the command are stored on the history list. For example, the ?? and SHH commands are quiet.

:INPUT These commands work more like macros, in that the result of evaluating the command is treated as a new line of input. The FIX command is an input command. The result is treated as a line; a single expression in EVAL-format should be returned as a list of the expression to EVAL.

Undoing

Note: This discussion only applies to undoing under the Exec or Debugger, and within the UNDOABLY macro; text and structure editors handle undoing differently.

The UNDO facility allows recording of destructive changes such that they can be played back to restore a previous state. There are two kinds of UNDOing: one is done by the Exec, the other is available for use in your code. Both methods share information about what kind of operations can be undone and where the changes are recorded.

Undoing in the Exec

UNDO *EventSpec* [Exec command]

The Exec's UNDO command is implemented by watching the evaluation of forms and requiring undoable operations in that evaluation to save enough information on the history list to reverse their side effects. The Exec simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible functions. The UNDO command works on itself the same way: it recovers the saved information and performs the corresponding inverses. Thus, UNDO is effective on itself, so that you can UNDO an UNDO, and UNDO that, etc.

Only when you attempt to undo an operation does the Exec check to see whether any information has been saved. If none has been saved, and you have specifically named the event you want undone, the Exec types nothing saved. (When you just type UNDO, the Exec only tries to undo the last operation.)

UNDO watches evaluation using CL:EVALHOOK (thus, calling CL:EVALHOOK cannot be undone). Each form given to EVAL is examined against the list LISPXFNS to see if it has a corresponding undoable version. If an undoable version of a call is found, it is called with the same arguments instead of the original. Therefore, before evaluating all subforms of your input, the Exec substitutes the corresponding undoable call for any destructive operation. For example, if you type (DEFUN FOO ...), undoable versions of the forms that set the definition into the symbol function cell are evaluated. FOO's function definition itself is not made undoable.

Undoing in Programs

There are two ways to make a program undoable. The simplest method is to wrap the program's form in the UNDOABLY macro. The other is to call undoable versions of destructive operations directly.

(**XCL:UNDOABLY** *&REST FORMS*) [Macro]

Executes the forms in *FORMS* using undoable versions of all destructive operations. This is done by "walking" (see WALKFORM) all of the *FORMS* and rewriting them to use the undoable versions of destructive operations (LISPXFNS makes the association).

(**STOP-UNDOABLY** *&REST FORMS*) [Macro]

Normally executes as PROGN; however, within an UNDOABLY form, explicitly causes *FORMS* not to be done undoably. Turns off rewriting of the *FORMS* to be undoable inside an UNDOABLY macro.

Undoable Versions of Common Functions

When efficiencyis a serious concern, you may need more control over the saving of undo information than that provided by the UNDOABLY macro.

To make a function undoable, you can simply substitute the corresponding undoable function in your program. When the undoable function is called, it will save the undo information in the current event on the history list.

Various operations, most notably SETF, have undoable versions. The following undoable macros are initially available:

UNDOABLY-POP	UNDOABLY-SET-SYMBOL
UNDOABLY-PUSH	UNDOABLY-MAKUNBOUND
UNDOABLY-PUSHNEW	UNDOABLY-FMAKUNBOUND
UNDOABLY-REMF	UNDOABLY-SETQ
UNDOABLY-ROTATEF	XCL:UNDOABLY-SETF
UNDOABLY-SHIFTF	UNDOABLY-PSETF
UNDOABLY-DECF	UNDOABLY-SETF-SYMBOL-FUNCTION
UNDOABLY-INCF	UNDOABLY-SETF-MACRO-FUNCTION

Note: Many destructive Common Lisp functions do not have undoable versions, e.g., CL:NREVERSE, CL:SORT, etc. You can see the current list of undoable functions on the association list LISPXFNS.

Modifying the UNDO Facility

You may want to extend the UNDO facility after creating a form whose side effects might be undoable, for instance a file renaming function.

You need to write an undoable version of the function. You can do this by explicitly saving previous state information, or by renaming calls in the function to their undoable equivalent. Undo information should be saved on the history list using IL:UNDOSAVE.

You must then hook the undoable version of the function into the undo facility. You do this by either using the IL:LISPXFNS association list, or in the case of a SETF modifier, on the IL:UNDOABLE-SETF-INVERSE property of the SETF function.

LISPXFNS [Variable]

Contains an association list that maps from destructive operations to their undoable form. Initially this list contains:

```
((CL:POP . UNDOABLY-POP)
 (CL:PSETF . NDOABLY-PSETF)
 (CL:PUSH . UNDOABLY-PUSH)
 (CL:PUSHNEW . UNDOABLY-PUSHNEW)
 ((CL:REMF) . UNDOABLY-REMF)
 (CL:ROTATEF . UNDOABLY-ROTATEF)
 (CL:SHIFTF . UNDOABLY-SHIFTF)
 (CL:DECF . UNDOABLY-DECF)
 (CL:INCF . UNDOABLY-INCF)
 (CL:SET . UNDOABLY-SET-SYMBOL)
 (CL:MAKUNBOUND . UNDOABLY-MAKUNBOUND)
 (CL:FMAKUNBOUND . UNDOABLY-FMAKUNBOUND)
 . . . plus the original Interlisp undo associations)
```

(**XCL:UNDOABLY-SETF** *PLACE VALUE* ...) [Macro]

Like CL:SETF but saves information so it may be undone. UNDOABLY-SETF uses undoable versions of the SETF function located on the UNDOABLE-SETF-INVERSE property of the function being SETFed. Initially these SETF names have such a property:

```
CL:SYMBOL-FUNCTION - UNDOABLY-SETF-SYMBOL-FUNCTION
CL:MACRO-FUNCTION - UNDOABLY-SETF-MACRO-FUNCTION
```

(**UNDOABLY-SETQ** *&REST FORMS*) [Function]

Typed-in SETQs (and SETFs on symbols) are made undoable by substituting a call to UNDOABLY-SETQ. UNDOABLY-SETQ operates like SETQ on lexical variables or those with dynamic bindings; it only saves information on the history list for changes to global, "top-level" values.

(**UNDOSAVE** *UNDOFORM HISTENTRY*) [Function]

Adds the undo information *UNDOFORM* to the SIDE property of the history event *HISTENTRY*. If there is no SIDE property, one is created. If the value of the SIDE property is NOSAVE, the information is not saved. *HISTENTRY* specifies an event. If *HISTENTRY*=NIL, the value of LISPXHIST is used. If both *HISTENTRY* and LISPXHIST are NIL, UNDOSAVE is a no-op.

The form of UNDOFORM is (FN . ARGS). Undoing is done by performing (APPLY (CAR UNDOFORM) (CDR UNDOFORM)).

\#UNDOSAVES [Variable]

The maximum number of UNDOFORMs to be saved for a single event. When the count of UNDOFORMs reaches this number, UNDOSAVE prints the message CONTINUE SAVING?, asking if you want to continue saving. If you answer NO or default, UNDOSAVE discards the previously saved information for this event, and makes NOSAVE be the value of the property SIDE, which disables any further saving for this event. If you answer YES, UNDOSAVE changes the count to -1, which is then never incremented, and continues saving. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need to be undone.

If \#UNDOSAVES is negative, then when the count reaches (ABS \#UNDOSAVES), UNDOSAVE simply stops saving without printing any messages or other interactions. \#UNDOSAVES = NIL is equivalent to \#UNDOSAVES = infinity. \#UNDOSAVES is initially NIL.

The configuration described here is very satisfactory. You pay a very small price for the ability to undo what you type in, since the interpreted evaluation is simply watched for destructive operations, or if you wish to protect yourself from malfunctioning in your own programs, you can explicitly call, or rewrite your program to explicitly call, undoable functions.

Undoing Out of Order

UNDOABLY-SETF operates undoably by saving (on the history list) the cell that is to be changed and its original contents. Undoing an UNDOABLY-SETF restores the saved contents.

This implementation can produce unexpected results when multiple modifications are made to the same piece of storage and then undone out of order. For example, if you type (SETF (CAR FOO) 1), followed by (SETF (CAR FOO) 2), then undo both events by undoing the most recent event first, then undoing the older event, FOO will be restored to its state before either event operated. However if you undo the first event, then the second event, (CAR FOO) will be 1, since this is what was in CAR of FOO before (UNDOABLY-SETF (CAR FOO) 2) was executed. Similarly, if you type

`(NCONC FOO '(1))`, followed by `(NCONC FOO '(2))`, undoing just `(NCONC FOO '(1))` will remove both `1` and `2` from `FOO`. The problem in both cases is that the two operations are not independent.

In general, operations are always independent if they affect different lists or different sublists of the same list. Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the right thing. However, undoing dependent operations out of order may not always have the predicted effect.

Format and Use of the History List

LISPXHISTORY [Variable]

The Exec currently uses one primary history list, `LISPXHISTORY` for the storing events.

The history list is in the form `(`*EVENTS EVENT# SIZE MOD*`)`, where *EVENTS* is a list of events with the most recent event first, *EVENT#* is the event number for the most recent event on *EVENTS*, *SIZE* is the the maximum length *EVENTS* is allowed to grow. *MOD* is is the maximum event number to use, after which event numbers roll over. *LISPXHISTORY* is initialized to `(NIL 0 100 1000)`.

The history list has a maximum length, called its time-slice. As new events occur, existing events are aged, and the oldest events are forgotten. The time-slice can be changed with the function `CHANGESLICE`. Larger time-slices enable longer memory spans, but tie up correspondingly greater amounts of storage. Since you seldom need really ancient history, a relatively small time-slice such as 30 events is usually adequate, although some users prefer to set the time-slice as large as 200 events.

Each individual event on *EVENTS* is a list of the form `(`*INPUT ID VALUE . PROPS*`)`. For Exec events, *ID* is a list `(`*EVENT-NUMBER EXEC-ID*`)`. The *EVENT-NUMBER* is the number of the event, while the *EXEC-ID* is a string that uniquely identifies the Exec. (The *EXEC-ID* is used to identify which events belong to the “same” Exec.) *VALUE* is the (first) value of the event. *PROPS* is a property list used to associate other information with the event (described below).

INPUT is the input sequence for the event. Normally, this is just the input that you type in. For an `APPLY`-format input this is a list consisting of two expressions; for an `EVAL`-format input, this is a list of just one expression; for an input entered as list of atoms, *INPUT* is simply that list. For example,

User Input	*INPUT* is:
`LIST(1 2)`	`(LIST (1 2))`
`(LIST 1 1)`	`((LIST 1 1))`
`DIR "{DSK}<LISPFILES>"`*cr*	`(DIR "{DSK}<LISPFILES>")`

If you type in an Exec command that executes other events (`REDO`, `USE`, etc.), several events might result. When there is more than one input, they are wrapped together into one invocation of the `DO-EVENTS` command.

The same convention is used for representing multiple inputs when a `USE` command involves sequential substitutions. For example, if you type `FBOUNDP(FOO)` and then `USE`

FIE FUM FOR FOO, the input sequence that will be constructed is DO-EVENTS (EVENT FBOUNDP (FIE)) (EVENT FBOUNDP (FUM)), which is the result of substituting FIE for FOO in (FBOUNDP (FOO)) concatenated with the result of substituting FUM for FOO in (FBOUNDP (FOO)).

PROPS is a property list of the form (PROPERTY$_1$ VALUE$_1$ PROPERTY$_2$ VALUE$_2$ ...), that can be used to associate arbitrary information with a particular event. Currently, the following properties are used by the Exec:

SIDE

A list of the side effects of the event. See UNDOSAVE.

LISPXPRINT

Used to record calls to EXEC-FORMAT, and printed by the ?? command.

Making or Changing an Exec

(**XCL:ADD-EXEC** *&KEY PROFILE REGION TTY ID*) [Function]

Creates a new process and window with an Exec running in it. *PROFILE* is the type of the Exec to be created (see below under XCL:SET-EXEC-TYPE). *REGION* optionally gives the shape and location of the window to be used. If not provided you will be prompted. *TTY* is a flag, which, if true, causes the tty to be given to the new Exec process. *ID* is a string identifier to use for events generated in this exec. *ID* defaults to the number given to the Exec process created.

(**XCL:EXEC** *&KEY WINDOW PROMPT COMMAND-TABLES ENVIRONMENT PROFILE TOP-LEVEL-P TITLE FUNCTION ID*) [Function]

This is the main entry to the Exec. The arguments are:

WINDOW defaults to the current TTY display stream, or can be provided a window in which the Exec will run.

PROMPT is the prompt to print.

COMMAND-TABLES is a list of hash-tables for looking up commands (e.g., *EXEC-COMMAND-TABLE* or *DEBUGGER-COMMAND-TABLE*).

ENVIRONMENT is a lexical environment used to evaluate things in.

READTABLE is the default readtable to use (defaults to the “Common Lisp” readtable).

PROFILE is a way to set the Exec’s type (see above, “Multiple Execs and the Exec’s Type”).

TOP-LEVEL-P is a boolean, which should be true if this Exec is at the top level (it’s NIL for debugger windows, etc).

TITLE is an identifying title for the window title of the Exec.

FUNCTION is a function used to actually evaluate events, default is EVAL-INPUT.

ID is a string identifier to use for events generated in this Exec. *ID* defaults to the number given to the Exec process.

XCL:*PER-EXEC-VARIABLES* [Variable]

A list of pairs of the form (*VAR INIT*). Each time an Exec is entered, the variables in *PER-EXEC-VARIABLES* are rebound to the value returned by evaluating *INIT*. The initial value of *PER-EXEC-VARIABLES* is:

```
((*PACKAGE* *PACKAGE*)
 (* *)
 (** **)
 (*** ***)
 (+ +)
 (++ ++)
 (+++ +++)
 (- -)
 (/ /)
 (// //)
 (/// ///)
 (HELPFLAG T)
 (*EVALHOOK* NIL)
 (*APPLYHOOK* nil)
 (*ERROR-OUPUT* *TERMINAL-IO*)
 (*READTABLE* *READTABLE*)
 (*package* *package*)
 (*eval-function* *eval-function*)
 (*exec-prompt* *exec-prompt*)
 (*debugger-prompt* *debugger-prompt*))
```

Most of these cause the values to be (re)bound to their current value in any inferior Exec, or to NIL, their value at the "top level".

XCL:*EVAL-FUNCTION* [Variable]

Bound to the function used by the Exec to evaluate input. Typically in an Interlisp Exec this is IL:EVAL, and in a Common Lisp Exec, CL:EVAL.

XCL:*EXEC-PROMPT* [Variable]

Bound to the string printed by the Exec as a prompt for input. Typically in an Interlisp Exec this is " ← ", and in a Common Lisp Exec, "> ".

XCL:*DEBUGGER-PROMPT* [Variable]

Bound to the string printed by the debugger Exec as a prompt for input. Typically in an Interlisp Exec this is " ← : ", and in a Common Lisp Exec, ": ".

(**XCL:EXEC-EVAL** *FORM &OPTIONAL ENVIRONMENT*) [Function]

Evaluates *FORM* (using EVAL) in the lexical environment *ENVIRONMENT* the same as though it were typed in to EXEC, i.e., the event is recorded, and the evaluation is made undoable by substituting the UNDOABLE-functions for the corresponding destructive functions. XCL:EXEC-EVAL returns the value(s) of the form, but does not print it, and does not reset the variables *, **, ***, etc.

(**XCL:EXEC-FORMAT** *CONTROL-STRING &REST ARGUMENTS*) [Function]

In addition to saving inputs and values, the Exec saves many system messages on the history list. For example, FILE CREATED ..., FN redefined, VAR reset, output of TIME, BREAKDOWN, ROOM, save their output on the history list, so that when ?? prints the event, the output is also printed. The function XCL:EXEC-FORMAT can be used in your code similarly. XCL:EXEC-FORMAT performs (APPLY #'CL:FORMAT *TERMINAL-IO* *CONTROL-STRING ARGUMENTS*) and also saves the format string and arguments on the history list associated with the current event.

(**XCL:SET-EXEC-TYPE** *NAME*) [Function]

Sets the type of the current Exec to that indicated by *NAME*. This can be used to set up the Exec to your liking. *NAME* may be an atom or string. Possible names are:

INTERLISP, IL	*READTABLE* INTERLISP *PACKAGE* INTERLISP XCL:*DEBUGGER-PROMPT* "←: " XCL:*EXEC-PROMPT* "←" XCL:*EVAL-FUNCTION* IL:EVAL
XEROX-COMMON-LISP, XCL	*READTABLE* XCL *PACKAGE* XCL-USER XCL:*DEBUGGER-PROMPT* ": " XCL:*EXEC-PROMPT* "> " XCL:*EVAL-FUNCTION* CL:EVAL
COMMON-LISP, CL	*READTABLE* LISP *PACKAGE* USER XCL:*DEBUGGER-PROMPT* ": " XCL:*EXEC-PROMPT* "> " XCL:*EVAL-FUNCTION* CL:EVAL
OLD-INTERLISP-T	*READTABLE* OLD-INTERLISP-T *PACKAGE* INTERLISP XCL:*DEBUGGER-PROMPT* "←: " XCL:*EXEC-PROMPT* ": " XCL:*EVAL-FUNCTION* IL:EVAL

(**XCL:SET-DEFAULT-EXEC-TYPE** *NAME*) [Function]

Like XCL:SET-EXEC-TYPE, but sets the type of Execs created by default, as from the background menu. Initially XCL. This can be used in your greet file to set default Execs to your liking.

Editing Exec Input

The Exec features an input editorwhich provides completion, spelling correction, help facility, and character-level editing. The implementation is borrowed from the Interlisp module TTYIN. This section describes the use of the TTYIN editor from the perspective of the Exec.

Editing Your Input

Some editing operations can be performed using any of several characters; characters that are interrupts will, of course, not be read, so several alternatives are given. The following characters may be used to edit your input:

CONTROL-A

BACKSPACE Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.

CONTROL-W Deletes a "word". Generally this means back to the last space or parenthesis.

CONTROL-Q Deletes the current line, or if the current line is blank, deletes the previous line.

CONTROL-R Refreshes the current line. Two in a row refreshes the whole buffer (when doing multiline input).

ESCAPE Tries to complete the current word from the spelling list USERWORDS. In the case of ambiguity, completes as far as is uniquely determined, or beeps.

UNDO key Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed; when typed in the middle of a line fills in the remaining text from the old line; when typed following CONTROL-Q or CONTROL-W restores what those commands erased.

CONTROL-X Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced.

If you are already at the end of the input and the expression is balanced except for lacking one or more right parentheses, CONTROL-X adds the required right parentheses to balance and returns.

During most kinds of input, lines are broken, if possible, so that no word straddles the end of the line. The pseudo-carriage return ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You will not get carriage returns in your strings unless you explicitly type them.

Using the Mouse

Editing with the mouse during TTYIN input is slightly different than with other modules. The mouse buttons are interpreted as follows during TTYIN input:

LEFT Moves the caret to where the cursor is pointing. As you hold down LEFT, the caret moves around with the cursor; after you let up, any type-in will be inserted at the new position.

MIDDLE
LEFT+RIGHT Like LEFT, but moves only to word boundaries.

RIGHT Deletes text from the caret to the cursor, either forward or backward. While you hold down RIGHT, the text to be deleted is inverted; when you let up, the text goes away. If you let up outside the scope of the text, nothing is deleted (this is how to cancel this operation).

If you hold down MOVE, COPY, SHIFT or CTRL while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. The selection is made by holding the appropriate key down while pressing the mouse buttons LEFT (to select a character) or MIDDLE (to select a word), and optionally extend the selection either left or right using RIGHT. While you are doing this, the caret does not move, but the selected text is highlighted in a manner indicating what is about to happen. When the selection is complete, release the mouse buttons and then lift up on MOVE/COPY/CTRL/SHIFT and the appropriate action will occur:

COPY
SHIFT The selected text is inserted as if it were typed. The text is highlighted with a broken underline during selection.

CTRL The selected text is deleted. The text is complemented during selection.

MOVE
CTRL+SHIFT Combines copy and delete. The selected text is moved to the caret.

You can cancel a selection in progress by pressing LEFT or MIDDLE as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing the UNDO key. This is the same key that retrieves the previous buffer when issued at the end of a line.

Editing Commands

A number of characters have special effects while typing to the Exec. Some of them merely move the caret inside the input stream. While caret positioning can often be done more conveniently with the mouse, some of the commands, such as the case changing commands, can be useful for modifying the input.

In the descriptions below, current word means the word the cursor is under, or if under a space, the previous word. Currently, parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands.

Most commands can be preceded by a numeric argument. A numeric argument can be a number or an escape. You enter the numeric argument by holding down the meta key and entering a number. You only need to hold down the meta key for the firs digit of the argument. Entering escape as a numeric argument means infinity.

Some commands also accept negative arguments, but some only look at the magnitude of the argument. Most of these commands are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands

Meta-BACKSPACE Backs up one (or n) characters.

Meta-SPACE Moves forward one (or n) characters.

Meta-^ Moves up one (or n) lines.

Meta-LINEFEED Moves down one (or n) lines.

Meta-(Moves back one (or n) words.

Meta-) Moves ahead one (or n) words.

Meta-tab Moves to end of line; with an argument moves to nth end of line; **Meta-Control-tab** goes to end of buffer.

Meta-Control-L Moves to start of line (or nth previous, or start of buffer).

Meta-{ Goes to start of buffer.

Meta-} Goes to end of buffer.

Meta-[Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under "Assorted Flags".)

Meta-] Moves to end of current list.

Meta-Sx Skips ahead to next (or nth) occurrence of character x, or rings the bell.

Meta-Bx Backward search.

Buffer Modification Commands

Meta-Zx Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command.

Meta-A
Meta-R Repeats the last S, B, or Z command, regardless of any intervening input.

Meta-K Kills the character under the cursor, or n chars starting at the cursor.

Meta-CR When the buffer is empty is the same as undo i.e. restores buffer's previous contents. Otherwise is just like a <cr> (except that it also terminates an insert). Thus, **Meta-CR Meta-CR** will repeat the previous input (as will undo<cr> without the meta key).

Meta-O Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

Meta-T Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle odd cases, such as tabs.

Meta-G Grabs the contents of the previous line from the cursor position onward. **Meta-**n **Meta-G** grabs the nth previous line.

Meta-L Puts the current word, or n words on line, in lower case. **Meta-**<escape> **Meta-L** puts the rest of the linein lower case; or if given at the end of line puts the entire line in lower case.

Meta-U Analogous to **Meta-L**, for putting word, line, or portion of line in upper case.

Meta-C Capitalizes. If you give it an argument, only the first word is capitalized; the rest are just lowercased.

Meta-Control-Q Deletes the current line. **Meta-**<escape> **Meta-Control-Q** deletes from the current cursor position to the end of the buffer. No other arguments are handled.

Meta-Control-W Deletes the current word, or the previous word if sitting on a space.

Miscellaneous Commands

Meta-P Prettyprints buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.

Meta-N Refreshes line. Same as **Control-R**. **Meta-**<escape> **Meta-N** refreshes the whole buffer; **Meta-**n **Meta-N** refreshes n lines. Cursor movement in TTYIN depends on TTYIN being the only source of output to the window; in some circumstances, you may need to refresh the line for best results.

Meta-Control-Y Gets an Interlisp Exec. **Meta-**<escape> **Meta-Control-Y** Gets an Interlisp Exec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for Interlisp, you can do **Meta-Control-L Meta-**<escape> **Meta-Control-Y** and give it to Lisp.

Meta-_ Adds the current word to the spelling list USERWORDS. With zero argument, removes word. See TTYINCOMPLETEFLG .

Useful Macros

If the event is considered short enough, the Exec command FIX will load the buffer with the event's input, rather than calling the structure editor. If you really wanted the Lisp editor for your fix, you can say FIX EVENT - |TTY:| .

?= Handler

Typing the characters ?=<cr> displays the arguments to the function currently in progress. Since TTYIN wants you to be able to continue editing the buffer after a ?=, it prints the arguments below your type-in and then puts the cursor back where it was when ?= was typed.

Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. All are initially set to T.

?ACTIVATEFLG [Variable]

If true, enables the feature whereby ? lists alternative completions from the current spelling list.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis, TTYIN will briefly move the cursor to the matching parenthesis, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you will never notice it).

USERWORDS [Variable]

USERWORDS contains words you mentioned recently: functions you have defined or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently edited a function, chances are good you may want to edit it again (typing "ED(xx$)") or type a call to it. If there is no completion for the current word from USERWORDS, or there is more than one possible completion, TTYIN beeps. If typed when not inside a word, Escape completes to the value of LASTWORD, i.e., the last thing you typed that the Exec noticed, except that Escape at the beginning of the line is left alone (it is an Old Interlisp Exec command).

If you really wanted to enter an escape, you can, of course, just quote it with a CONTROL-V, like you can other control characters.

You may explicitly add words to USERWORDS yourself that would not get there otherwise. To make this convenient online the edit command [←] means "add the current atom to USERWORDS" (you might think of the command as pointing out this atom). For example, you might be entering a function definition and want to point to one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from USERWORDS.

Note that this feature loses some of its value if the spelling list is too long, if there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list USERWORDS keeps the temporary section (which is where everything goes initially unless you say otherwise) limited to \#USERWORDS atoms, initially 100. Words fall off the end if they haven't been used (they are used if FIXSPELL corrects to one, or you use <escape> to complete one).

Old Interlisp T compatibility

The Old Interlisp exec contains a few extra Exec commands not listed above. They are explained here.

In addition to the normal Event addresses you can also specify the following Event addresses:

= Specifies that the next object is to be searched for in the values of events, instead of the inputs

SUCHTHAT *PRED* Specifies an event for which the function *PRED* returns true. *PRED* should be a function of two arguments, the input portion of the event, and the event itself.

PAT Any other event address command specifies an event whose input contains an expression that matches *PAT*. When multiple Execs are active, all events are searched, no matter which Exec they belong to. The pattern can be a simple symbol, or a more complex search pattern.

Significant Changes in MEDLEY Rele[redacted]ase

There are two major differences between the Medley release and older versions of the system:

- SETQ does not interact with the File Manager. In older releases (Koto, etc.), when you typed in (SETQ FOO *some-new-value*) the executive responded with (FOO reset) and the file manager was told that FOO's value had changed. Files containing FOO were marked for cleanup, if none existed you were prompted for one when you typed (FILES?).

  This is still the case in the Old Interlisp executive but not in any of the others. If you are setting a variable that is significant to a program and you want to save it on a file, you should use the Common Lisp macro CL:DEFPARAMETER instead of SETQ. This will give the symbol a definition of type VARIABLES (instead of VARS), and it will be noticed by the File Manager. Subseqent changes to the variable must be done by another call to CL:DEFPARAMETER or by editing it using ED (not DV).

- The following functions and variables are only available in the Old Interlisp Exec: LISPX, USEREXEC, LISPXEVAL, READBUF, (READLINE), (LISPXREAD), (LISPXREADP), (LISPXUNREAD), (PROMPTCHAR), (HISTORYSAVE), (LISPXSTOREVALUE), (LISPXFIND), (HISTORYFIND), (HISROTYMATCH), (ENTRY), (UNDOSAVE), #UNDOSAVES, (NEW/FN), (LISPX/), (UNDOLISPX), (UNDOLISPX1), and (PRINTHISTORY).

  The function USEREXEC invokes an old-style executive, but uses the package and readtable of its caller. Callers of LISPXEVAL should use EXEC-EVAL instead.

14. ERRORS AND DEBUGGING

Occasionally, while a program is running, an error occurs which stops the computation. Errors can be caused in different ways. A coding mistake may have caused the wrong arguments to be passed to a function, or caused the function to attempt something illegal. For example, PLUS will cause an error if its arguments are not numbers. It is also possible to interrupt a computation by typing one of the "interrupt characters," such as Control-D or Control-E (Medley interrupt characters are listed in Chapter 30). Finally, you can specify that certain functions automatically cause an error whenever they are entered (see Chapter 15). This facilitates debugging by allowing you to examine the context within the computation.

When an error occurs, the system can either reset and unwind the stack, or go into a "break", and attempt to debug the program. You can modify the mechanism that decides whether to unwind the stack or break, and is described in the Controlling When to Break section in this chapter. Within a break, Medley offers an extensive set of "break commands".

This chapter explains what happens when errors occur. It also tells you how to handle program errors using breaks and break commands. The debugging capabilities of the break window facility are described, as well as the variables that control its operation. Finally, advanced facilities for modifying and extending the error mechanism are presented.

Breaks

One of the most useful debugging facilities in Medley is the ability to put the system into a "break", stopping a computation at any point, allowing you to interrogate the state of the world and affect the course of the computation. When a break occurs, a "break window" (see the Break Windows section below) is brought up near the TTY window of the broken process. The break window looks like a top-level executive window, except that the prompt character is "**:**" instead of "←" as in the top-level executive. A break saves the environment where the break occurred, so that you may evaluate variables and expressions in the borken environment. In addition, the break program recognizes a number of useful "break commands", providing an easy way to interrogate the state of the broken computation.

Breaks may be entered in several ways. Some interrupt characters (Chapter 30) automatically cause a break whenever you type them. Function errors may also cause a break, depending on the depth of the computation (see Controlling When to Break below). Finally, Medley provides facilities which make it easy to "break" suspect functions so that they always cause a break whenever they are entered.

Within a break you have access to all of the power of Medley; you can do anything you can do at the top-level executive. For example, you can evaluate an expression, call the editor, change the function, and evaluate the expression again, all without leaving the break. You can also type in commands like REDO, and UNDO (Chapter 13), to redo or undo previously executed events, including break commands.

Similarly, you can prettyprint functions, define new functions or redefine old ones, load a file, compile functions, time a computation, etc. In addition, you can examine the stack (see Chapter 11), and even force a return back to some higher function via the functions RETFROM or RETEVAL.

Once a break occurs, *you* are in complete control of the flow of the computation, and the computation will not proceed without specific instruction from you. If you type in an expression whose evaluation causes an error, the break is maintained. Similarly if you abort a computation initiated from within the break (by typing Control-E), the break is maintained. Only if you give one of the commands that exits from the break, or evaluates a form which does a RETFROM or RETEVAL out of BREAK1, will the computation continue. Also, BREAK1 does not “turn off” Control-D, so a Control-D will force an immediate return to the top level.

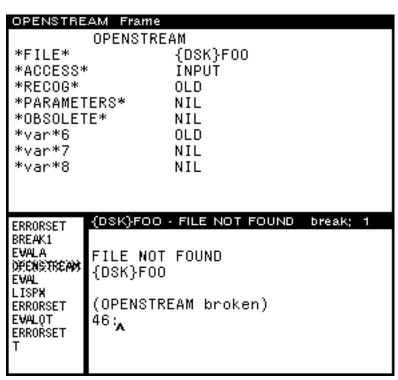
Break Windows

When a break occurs, a break window is brought up near the TTY window of the borken process and the terminal stream switched to it. The title of the break window is changed to the name of the broken function and the reason for the break. If a break occurs under a previous break, a new break window is created.

If a break is caused by a storage full error, the display break package will not try to open a new break window, since this would cause an infinite loop.

While in a break window, clicking the middle button brings up a menu of break commands: EVAL, EDIT, revert, ↑, OK, BT, BT!, and ?=. Clicking on these commands is equivalent to typing the corresponding break commandm, except BT and BT! which behave differently from the typed-in commands (see Break Commands below).

The BT and BT! menu commands bring up a backtrace menu beside the break window showing the frames on the stack. BT shows frames for which REALFRAMEP is T; BT! shows all frames. When one of the frames is selected from the backtrace menu, it is grayed and the function name and the variables bound in that frame (including local variables and PROG variables) are printed in the “backtrace frame window.” If the left button is used for the selection, only named variables are printed. If the middle button is used, all variables are printed (variables without names appear as *var* *N*). The “backtrace frame” window is an inspect window (see Chapter 26). In this window, the left button is used to select the name of the function, the names of the variables or the values of the variables. For example, below is a picture of a break window with a backtrace menu created by BT. The OPENSTREAM stack frame has been selected, so its variables are shown in an inspect window on top of the break window:



After selecting an item, the middle button brings up a menu of commands that apply to the selected item. If the function name is selected, you are given a choice of editing the function or seeing the compiled code with INSPECTCODE (Chapter 26). If you edit the function in this way, the editor is called in the broken process, so variables evaluated in the editor are in the broken process.

If a variable name is selected, the command SET is offered. Selecting SET will READ a value and set the selected to the value read.

Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. Exercise caution in setting variables in other than your own code.

If a value is selected, the inspector is called on the selected value.

The internal break variable LASTPOS (see the section below) is set to the selected backtrace menu frame so that the normal break commands EDIT, revert, and ?= work on the currently selected frame. The commands EVAL, revert, ↑, OK, and ?= in the break menu cause the corresponding commands to be "typed in." This means that these break commands will not have the intended effect if characters have already been typed in. The typed-in break commands BT, BTV, etc. use the value of LASTPOS to determine where to start listing the stack, so selecting a stack frame name in the backtrace menu affects these commands.

Break Commands

The basic function of the break package is BREAK1. BREAK1 is just another Interlisp function, not a special system feature like the interpreter or the garbage collector. It has arguments, and returns a value, the same as any other function. For more information on the function BREAK1, see Creating Breaks with BREAK1 below.

The value returned by BREAK1 is called "the value of the break." You can specify this value explicitly by using the RETURN break command (see below). But in most cases, the value of a break is given implicitly, via a GO or OK command, and is the result of evaluating "the break expression." The break expression, stored in the variable BRKEXP, is an expression equivalent to the computation that would have taken place had no break occurred. For example, if you break on the function FOO, the break expression is the body of the definition of FOO. When you type OK or GO, the body of FOO is evaluated, and its value returned as the value of the break, i.e., to whatever function called FOO. BRKEXP is set up by the function that created the call to BREAK1. For functions broken with BREAK or TRACE, BRKEXP is equivalent to the body of the definition of the broken function (see Chapter 15). For functions broken with BREAKIN, using BEFORE or AFTER, BRKEXP is NIL. For BREAKIN AROUND, BRKEXP is the indicated expression (see Chapter 15).

BREAK1 recognizes a large set of break commands. These are typed in *without* parentheses. In order to facilitate debugging of programs that perform input operations, the carriage return that is typed to complete the GO, OK, EVAL, etc. commands is discarded by BREAK1, so that it will not be part of the input stream after the break.

GO [Break Command]

Evaluates BRKEXP, prints its value, and returns it as the value of the break. Releases the break and allows the computation to proceed.

OK [Break Command]

Same as GO except that the value of BRKEXP is not printed.

EVAL [Break Command]

Same as OK except that the break is maintained after the evaluation. The value of EVAL is bound to the local variable !VALUE, which you can interrogate. Typing GO or OK following EVAL will not cause BRKEXP to be reevaluated, but simply returns the value of !VALUE as the value of the break. Typing another EVAL will cause reevaluation. EVAL is useful when you are not sure whether the break will produce the correct value and want to examine it before continuing with the computation.

RETURN *FORM* [Break Command]

FORM is evaluated, and returned as the value of the break. For example, one could use the EVAL command and follow this with RETURN (REVERSE !VALUE).

↑ [Break Command]

Calls ERROR! and aborts the break, making it "go away" without returning a value. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the GO, OK, EVAL, and RETURN commands, maintain the break.

The following four commands refer to "the broken function", whose name is stored in the BREAK1 argument BRKFN.

!GO [Break Command]

The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited with the value printed.

!OK [Break Command]

The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited without the value printed.

UB [Break Command]

Unbreaks the broken function.

@ [Break Command]

Resets the variable LASTPOS, which establishes a context for the commands ?=, ARGS, BT, BTV, BTV*, EDIT, and IN? described below. LASTPOS is the position of a function call on the stack. It is initialized to the function just before the call to BREAK1, i.e., (STKNTH -1 'BREAK1).

When control passes from BREAK1, e.g. as a result of an EVAL, OK, GO, REVERT, ↑ command, or via a RETFROM or RETEVAL you type in, (RELSTK LASTPOS) is executed to release this stack pointer.

@ treats the rest of the teletype line as its argument(s). It first resets LASTPOS to (STKNTH -1 'BREAK1) and then for each atom on the line, @ searches down the stack for a call to that atom. The following atoms are treated specially:

@ Do not reset LASTPOS to (STKNTH -1 'BREAK1) but leave it as it was, and continue searching from that point.

a number *N* If negative, move LASTPOS down the stack *N* frames. If positive, move LASTPOS up the stack *N* frames.

/ The next atom on the line (which should be a number) specifies that the *previous* atom should be searched for that many times. For example, "@ FOO / 3" is equivalent to "@ FOO FOO FOO".

= Resets LASTPOS to the *value* of the next expression, e.g., if the value of FOO is a stack pointer, "@ = FOO FIE" will search for FIE in the environment specified by (the value of) FOO.

For example, if the push-down stack looks like:

```
[9]    BREAK1
[8]    FOO
[7]    COND
[6]    FIE
[5]    COND
[4]    FIE
[3]    COND
[2]    FIE
[1]    FUM
```

then "@ FIE COND" will set LASTPOS to the position corresponding to *[5]*; "@ @ COND" will then set LASTPOS to *[3]*; and "@ FIE / 3 - 1" to *[1]*.

If @ cannot successfully complete a search for function *FN*, it searches the stack again from that point looking for a call to a function whose name is a possible misspelling of *FN* (see spelling correction in Chapter 20). If the search is still unsuccessful, @ types (*FN* NOT FOUND), and then aborts.

When @ finishes, it types the name of the function at LASTPOS, i.e., (STKNAME LASTPOS).

@ can be used on BRKCOMS (see Creating Breaks with BREAK1 below). In this case, the *next* command on BRKCOMS is treated the same as the rest of the teletype line.

?= [Break Command]

This is a multi-purpose command. Its most common use is to interrogate the value(s) of the arguments of the broken function. For example, if FOO has three arguments (X Y Z), then typing ?= to a break on FOO will produce:

:?=
X = *value of* X
Y = *value of* Y
Z = *value of* Z
:

?= operates on the rest of the teletype line as its arguments. If the line is empty, as in the above case, it operates on all of the arguments of the broken function. If the you type ?= X (CAR Y), you will see the value of X, and the value of (CAR Y). The difference between using ?= and typing X and (CAR Y) directly to BREAK1 is that ?= evaluates its inputs as of the stack frame LASTPOS, i.e., it uses STKEVAL. This provides a way of examining variables or performing computations *as of a particular point on the stack.* For example, @ FOO / 2 followed by ?= X will allow you to examine the value of X in the previous call to FOO, etc.

?= also recognizes numbers as referring to the correspondingly numbered argument, i.e., it uses STKARG in this case. Thus

```
:@ FIE
FIE
:?= 2
```

will print the name and value of the second argument of FIE.

?= can also be used on BRKCOMS (see Creating Breaks with BREAK1 below), in which case the next command on BRKCOMS is treated as the rest of the teletype line. For example, if BRKCOMS is (EVAL ?= (X Y) GO), BRKEXP is evaluated, the values of X and Y printed, and then the function exited with its value being printed.

?= prints variable values using the function SHOWPRINT (see Chapter 25), so that if SYSPRETTYFLG = T, the value is prettyprinted.

?= is a universal mnemonic for displaying argument names and their corresponding values. In addition to being a break command, ?= is an edit macro that prints the argument names and values for the current expression (see Chapter 16), and a read macro (actually ? is the read macro character) which does the same for the current level list being read.

PB [Break Command]

Prints the bindings of a given variable. Similar to ?=, except ascends the stack starting from LASTPOS, and, for each frame in which the given variable is bound, prints the frame name and value of the variable (with PRINTLEVEL reset to (2 . 3)), e.g.

```
:PB FOO
@  FN1:  3
@  FN2:  10
@  TOP:  NOBIND
```

PB is also a programmer's assistant command (see Chapter 13) that can be used when not in a break. PB is implemented via the function PRINTBINDINGS.

BT [Break Command]

Prints a backtrace of function names starting at LASTPOS. The value of LASTPOS is changed by selecting an item from the backtrace menu (see the Break Window Variables section below) or by the @ command. The several nested calls in system packages such as break, edit, and the top level executive appear as the single entries **BREAK**, **EDITOR**, and **TOP** respectively.

BTV [Break Command]

Prints a backtrace of function names *with* variables beginning at LASTPOS.

The value of each variable is printed with the function SHOWPRINT (see Chapter 25), so that if SYSPRETTYFLG = T, the value is prettyprinted.

BTV+ [Break Command]

Same as BTV except also prints local variables and arguments to SUBRs.

BTV* [Break Command]

Same as BTV except prints arguments to local variables.

BTV! [Break Command]

Same as BTV except prints *everything* on the stack.

BT, BTV, BTV+, BTV*, and BTV! all take optional functional arguments. Use these arguments to choose functions to be *skipped* on the backtrace. As the backtrace scans down the stack, the name of each stack frame is passed to each of the arguments of the backtrace command. If any of these functions returns a non-NIL value, then that frame is skipped, and not shown in the backtrace. For example, BT EXPRP will skip all functions definied by expr definitions, BTV (LAMBDA (X) (NOT (MEMB X FOOFNS))) will skip all but those functions on FOOFNS. If used on BRKCOMS (see Creating Breaks with BREAK1 below) the functional argument is no longer optional, i.e., the next element on BRKCOMS must either be a list of functional arguments, or NIL if no functional argument is to be applied.

For BT, BTV, BTV+, BTV*, and BTV!, if Control-P is used to change a printlevel during the backtrace, the printlevel is restored after the backtrace is completed.

The value of BREAKDELIMITER, initially the carriage return character, is printed to delimit the output of ?= and backtrace commands. You can reset it (e.g. to a comma) for more linear output.

ARGS [Break Command]

Prints the names of the variables bound at LASTPOS, i.e., (VARIABLES LASTPOS) (see Chapter 11). For most cases, these are the arguments to the function entered at that position, i.e., (ARGLIST (STKNAME LASTPOS)).

REVERT [Break Command]

Goes back to position LASTPOS on stack and reenters the function called at that point with the arguments found on the stack. If the function is not already broken, REVERT first breaks it, and then unbreaks it after it is reentered.

REVERT can be given the position using the conventions described for @, e.g., REVERT FOO -1 is equivalent to @ FOO -1 followed by REVERT.

REVERT is useful for restarting a computation in the situation where a bug is discovered at some point *below* where the problem actually occurred. REVERT essentially says “go back there and start over in a break.” REVERT will work correctly if the names or arguments to the function, or even its function type, have been changed.

ORIGINAL [Break Command]

For use in conjunction with BREAKMACROS (see Creating Breaks with BREAK1 below). Form is (ORIGINAL . *COMS*). *COMS* are executed without regard for BREAKMACROS. Useful for redefining a break command in terms of itself.

EDIT [Break Command]

Designed for use in conjunction with breaks caused by errors. Facilitates editing the expression causing the break:

```
NON-NUMERIC ARG
NIL
(IPLUS BROKEN)
:EDIT
IN FOO...
(IPLUS X Z)
EDIT
*(3 Y)
*OK
FOO
:
```

and you can continue by typing OK, EVAL, etc.

This command is very simple conceptually, but its implementation is complicated by all of the exceptional cases involving interactions with compiled functions, breaks on user functions, error breaks, breaks within breaks, et al. Therefore, we shall give the following simplified explanation which will account for 90% of the situations arising in actual usage. For those others, EDIT will print an appropriate failure message and return to the break.

EDIT begins by searching up the stack beginning at LASTPOS (set by @ command, initially position of the break) looking for a form, i.e., an internal call to EVAL. Then EDIT continues from that point looking for a call to an interpreted function, or to EVAL. It then calls the editor on either the EXPR or the argument to EVAL in such a way as to look for an expression EQ to the form that it first found. It then prints the form, and permits interactive editing to begin. You can then type successive 0's to the editor to see the chain of superforms for this computation.

If you exit from the edit with an OK, the break expression is reset, if possible, so that you can continue with the computation by simply typing OK. (Evaluating the new BRKEXP will involve reevaluating the form that causes the break, so that if (PUTD (QUOTE (FOO)) *BIG-COMPUTATION*) were handled by EDIT, *BIG-COMPUTATION* would be reevaluated.) However, in some situations, the break expression cannot be reset. For example, if a compiled function FOO incorrectly called PUTD and caused the error Arg not atom followed by a break on PUTD, EDIT might be able to find the form headed by FOO, and also find *that* form in some higher interpreted function. But after you corrected the problem in the FOO-form, if any, you would still not have informed EDIT what to do about the immediate problem, i.e., the incorrect call to PUTD. However, if FOO were *interpreted*, EDIT would find the PUTD form itself, so that when you corrected that form, EDIT could use the new corrected form to reset the break expression.

IN? [Break Command]

Similar to EDIT, but just prints parent form, and superform, but does not call the editor, e.g.,

```
ATTEMPT TO RPLAC NIL
T
(RPLACD BROKEN)
:IN?
FOO:  (RPLACD X Z)
```

Although EDIT and IN? were designed for error breaks, they can also be useful for user breaks. For example, if upon reaching a break on his function FOO, you determine that there is a problem in the *call* to FOO, you can edit the calling form and reset the break expression with one operation by using EDIT.

Controlling When to Break

When an error occurs, the system has to decide whether to reset and unwind the stack, or go into a break. In the middle of a complex computation, it is usually helpful to go into a break, so that you may examine the state of the computation. However, if the computation has only proceeded a little when the error occurs, such as when you mistype a function name, you would normally just terminate a break, and it would be more convenient for the system to simply cause an error and unwind the stack in this situatuation. The decision over whether or not to induce a break depends on the depth of computation, and the amount of time invested in the computation. The actual algorithm is described in detail below; suffice it to say that the parameters affecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

(**BREAKCHECK** *ERRORPOS ERXN*) [Function]

BREAKCHECK is called by the error routine to decide whether or not to induce a break when a error occurs. *ERRORPOS* is the stack position at which the error occurred; *ERXN* is the error number. Returns T if a break should occur; NIL otherwise.

BREAKCHECK returns T (and a break occurs) if the “computation depth” is greater than or equal to HELPDEPTH. HELPDEPTH is initially set to 7, arrived at empirically by taking into account the overhead due to LISPX or BREAK.

If the depth of the computation is less than HELPDEPTH, BREAKCHECK next calculates the length of time spent in the computation. If this time is greater than HELPTIME milliseconds, initially set to 1000, then BREAKCHECK returns T (and a break occurs), otherwise NIL.

BREAKCHECK determines the “computation depth” by searching back up the stack looking for an ERRORSET frame (ERRORSETs indicate how far back unwinding is to take place when an error occurs, see the Catching Errors section below). At the same time, it counts the number of internal calls to EVAL. As soon as the number of calls to EVAL exceeds HELPDEPTH, BREAKCHECK immediately stops searching for an ERRORSET and returns T. Otherwise, BREAKCHECK continues searching until either an ERRORSET is found or the top of the stack is reached. (If the second argument to ERRORSET is INTERNAL, the ERRORSET is ignored by BREAKCHECK during this search.) BREAKCHECK then counts the number of function calls between the error and the last ERRORSET, or the top of the stack.

The number of function calls plus the number of calls to EVAL (already counted) is used as the "computation depth".

BREAKCHECK determines the computation time by subtracting the value of the variable HELPCLOCK from the value of (CLOCK 2), the number of milliseconds of compute time (see Chapter 12). HELPCLOCK is rebound to the current value of (CLOCK 2) for each computation typed in to LISPX or to a break. The time criterion for breaking can be suppressed by setting HELPTIME to NIL (or a very big number), or by setting HELPCLOCK to NIL. Setting HELPCLOCK to NIL will not have any effect beyond the current computation, because HELPCLOCK is rebound for each computation typed in to LISPX and BREAK.

You can suppress all error breaks by setting the top level binding of the variable HELPFLAG to NIL using SETTOPVAL (HELPFLAG is bound as a local variable in LISPX, and reset to the global value of HELPFLAG on every LISPX line, so just SETQing it will not work.) If HELPFLAG = T (the initial value), the decision whether to cause an error or break is decided based on the computation time and the computation depth, as described above. Finally, if HELPFLAG = BREAK!, a break will always occur following an error.

Break Window Variables

The appearance and use of break windows is controlled by the following variables:

(**WBREAK** *ONFLG*) [Function]

If *ONFLG* is non-NIL, break windows and trace windows are enabled. If *ONFLG* is NIL, break windows are disabled (break windows do not appear, but the executive prompt is changed to "**:**" to indicate that the system is in a break). WBREAK returns T if break windows are currently enabled; NIL otherwise.

MaxBkMenuWidth [Variable]
MaxBkMenuHeight [Variable]

The variables MaxBkMenuWidth (default 125) and MaxBkMenuHeight (default 300) control the maximum size of the backtrace menu. If this menu is too small to contain all of the frames in the backtrace, it is made scrollable in both vertical and horizontal directions.

AUTOBACKTRACEFLG [Variable]

This variable controls when and what kind of backtrace menu is automatically brought up. The value of AUTOBACKTRACEFLG can be one of the following:

NIL The backtrace menu is not automatically brought up (the default).

T On error breaks the BT menu is brought up.

BT! On error breaks the BT! menu is brought up.

ALWAYS The BT menu is brought up on both error breaks and user breaks (calls to functions broken by BREAK).

ALWAYS! On both error breaks and user breaks the BT! menu is brought up.

BACKTRACEFONT [Variable]

The backtrace menu is printed in the font BACKTRACEFONT.

CLOSEBREAKWINDOWFLG [Variable]

The system normally closes break windows after the break is exited. If CLOSEBREAKWINDOWFLG is NIL, break windows will not be closed on exit. In this case, you must close all break windows.

BREAKREGIONSPEC [Variable]

Break windows are positioned near the TTY window of the broken process, as determined by the variable BREAKREGIONSPEC. The value of this variable is a region (see Chapter 27) whose LEFT and BOTTOM fields are an offset from the LEFT and BOTTOM of the TTY window. The WIDTH and HEIGHT fields of BREAKREGIONSPEC determine the size of the break window.

TRACEWINDOW [Variable]

The trace window, TRACEWINDOW, is used for tracing functions. It is brought up when the first tracing occurs and stays up until you close it. TRACEWINDOW can be set to a particular window to cause the tracing formation to print there.

TRACEREGION [Variable]

The trace window is first created in the region TRACEREGION.

Creating Breaks with BREAK1

The basic function of the break package is BREAK1, which creates a break. A break appears to be a regular executive, with the prompt "**:**", but BREAK1 also detects and interpretes break commands (see the Break Commands section above).

(**BREAK1** *BRKEXP BRKWHEN BRKFN BRKCOMS BRKTYPE ERRORN*) [NLambda Function]

If *BRKWHEN* (evaluated) is non-NIL, a break occurs and commands are then taken from *BRKCOMS* or the terminal and interpreted. All inputs not recognized by BREAK1 are simply passed on to the programmer's assistant.

If *BRKWHEN* is NIL, *BRKEXP* is evaluated and returned as the value of BREAK1, without causing a break.

When a break occurs, if *ERRORN* is a list whose CAR is a number, ERRORMESS (see the Signalling Errors section below) is called to print an identifying message. If *ERRORN* is a list whose CAR is not a number, ERRORMESS1 (see the Signalling Errors section below) is called. Otherwise, no preliminary message is printed. Following this, the message (*BRKFN* broken) is printed.

Since BREAK1 itself calls functions, when one of these is broken, an infinite loop would occur. BREAK1 detects this situation, and prints Break within a break on *FN*, and then simply calls the function without going into a break.

The commands GO, !GO, OK, !OK, RETURN and ↑ are the only ways to leave BREAK1. The command EVAL causes *BRKEXP* to be evaluated, and saves the value on the variable !VALUE. Other commands can be defined for BREAK1 via BREAKMACROS (below).

BRKTYPE is NIL for user breaks, INTERRUPT for Control-H breaks, and ERRORX for error breaks. For breaks when *BRKTYPE* is not NIL, BREAK1 will clear and save the input buffer. If the break returns a value (i.e., is not aborted via ↑ or Control-D) the input buffer is restored.

The fourth argument to BREAK1 is *BRKCOMS*, a list of break commands that BREAK1 interprets and executes as though they were keyboard input. One can think of *BRKCOMS* as another input file which always has priority over the keyboard. Whenever *BRKCOMS* = NIL, BREAK1 reads its next command from the keyboard. Whenever *BRKCOMS* is non-NIL, BREAK1 takes (CAR BRKCOMS) as its next command and sets *BRKCOMS* to (CDR BRKCOMS). For example, suppose you wished to see the value of the variable X *after* a function was evaluated. You could set up a break with *BRKCOMS* = (EVAL (PRINT X) OK), which would have the desired effect. If *BRKCOMS* is non-NIL, the value of a break command is not printed. If you desire to see a value, you must print it yourself, as in the above example. The function TRACE (see Chapter 15) uses *BRKCOMS*: it sets up a break with two commands; the first one prints the arguments of the function, or whatever you specify, and the second is the command GO, which causes the function to be evaluated and its value printed.

Note: If an error occurs while interpreting the *BRKCOMS* commands, *BRKCOMS* is set to NIL, and a full interactive break occurs.

The break package has a facility for redirecting ouput to a file. All output resulting from *BRKCOMS* is output to the value of the variable BRKFILE, which should be the name of an open file. Output due to user type-in is not affected, and will always go to the terminal. BRKFILE is initially T.

BREAKMACROS [Variable]

BREAKMACROS is a list of the form ((*NAME*$_1$ *COM1*$_1$ ... *COM1*$_n$)(*NAME*$_2$ *COM2*$_1$ ... *COM2*$_n$) ...). Whenever an atomic command is given to BREAK1, it first searches the list BREAKMACROS for the command. If the command is equal to *NAME*$_i$, BREAK1 simply appends the corresponding commands to the front of *BRKCOMS*, and goes on. If the command is not found on BREAKMACROS, BREAK1 then checks to see if it is one of the built in commands, and finally, treats it as a function or variable as before.

If the command is not the name of a defined function, bound variable, or LISPX command, BREAK1 will attempt spelling correction using BREAKCOMSLST as a spelling list. If spelling correction is unsuccessful, BREAK1 will go ahead and call LISPX anyway, since the atom may also be a misspelled history command.

For example, the command ARGS could be defined by including on BREAKMACROS the form:

```
(ARGS (PRINT (VARIABLES LASTPOS T)))
```

(**BREAKREAD** *TYPE*) [Function]

Useful within BREAKMACROS for reading arguments. If BRKCOMS is non-NIL (the command in which the call to BREAKREAD appears was not typed in), returns the next break command from BRKCOMS, and sets BRKCOMS to (CDR BRKCOMS).

If BRKCOMS is NIL (the command was typed in), then BREAKREAD returns either the rest of the commands on the line as a list (if *TYPE* = LINE) or just the next command on the line (if *TYPE* is not LINE).

For example, the BT command is defined as (BAKTRACE LASTPOS NIL (BREAKREAD 'LINE) 0 T). Thus, if you type BT, the third argument to BAKTRACE is NIL. If you type BT SUBRP, the third argument is (SUBRP) .

BREAKRESETFORMS [Variable]

If you are developing programs that change the way a user and Medley normally interact (e.g., change or disable the interrupt or line-editing characters, turn off echoing, etc.), debugging them by breaking or tracing may be difficult, because Medley might be in a "funny" state at the time of the break. BREAKRESETFORMS is designed to solve this problem. You put in BREAKRESETFORMS expressions suitable for use in conjunction with RESETFORM or RESETSAVE (see Changing and Restoring System State below). When a break occurs, BREAK1 evaluates each expression on BREAKRESETFORMS *before* any interaction with the terminal, and saves the values. When the break expression is evaluated via an EVAL, OK, or GO, BREAK1 first restores the state of the system with respect to the various expressions on BREAKRESETFORMS. When control returns to BREAK1, the expressions on BREAKRESETFORMS are *again* evaluated, and their values saved. When the break is exited with an OK, GO, RETURN, or ↑ command, by typing Control-D, or by a RETFROM or RETEVAL you type in, BREAK1 again restores state. Thus the net effect is to make the break invisible with respect to your programs, but nevertheless allow you to interact in the break in the normal fashion.

All user type-in is scanned to make the operations undoable, as described in Chapter 13. At this point, RETFROMs and RETEVALs are also noticed. However, if you type in an expression which calls a function that then does a RETFROM, this RETFROM will not be noticed, and the effects of BREAKRESETFORMS will *not* be reversed.

As mentioned earlier, BREAK1 detects "Break within a break" situations, and avoids infinite loops. If the loop occurs because of an error, BREAK1 simply rebinds BREAKRESETFORMS to NIL, and calls HELP. This situation most frequently occurs when there is a bug in a function called by BREAKRESETFORMS.

SETQ expressions can also be included on BREAKRESETFORMS for saving and restoring system parameters, e.g. (SETQ LISPXHISTORY NIL), (SETQ DWIMFLG NIL), etc. These are handled specially by BREAK1 in that the current value of the variable is saved before the SETQ is executed, and upon restoration, the variable is set back to this value.

Signalling Errors

With the Medley release, Interlisp errors use the Xerox Common Lisp (XCL) error system. Most of the functions still exist for compatibility with previous releases, but the underlying machinery has changed. There are some incompatible differences, especially with respect to error numbers. All errors are now handled by signalling an object of type XCL:CONDITION. This means the error numbers generated are different from the old Interlisp method of registered numbers for well-known errors and error messages for all other errors. The mapping from Interlisp erors to Lisp error conditions is listed in the Error List sections below. The obsolete error numbers still generate error messages, but they are useless.

(**ERRORX** *ERXM*) [Function]

Calls CL:ERROR after first converting *ERXM* into a condition. If *ERXM* is NIL the value of *LAST-CONDITION* is used. If *ERXM* is an Interlisp error descriptor, it is first converted to a condition. If *ERXM* is already a condition, it is passed along unchanged. ERRORX also sets a proceed case for XCL:PROCEED, which will attempt to re-evaluate the caller of ERRORX, much as OK did in older versions of the break package.

(**ERROR** $MESS_1$ $MESS_2$ *NOBREAK*) [Function]

Prints $MESS_1$ (using PRIN1), followed by a space if $MESS_1$ is an atom, otherwise a carriage return. Then $MESS_2$ is printed (using PRIN1 if $MESS_2$ is a string; otherwise PRINT). For example, (ERROR "NON-NUMERIC ARG" T) prints

```
NON-NUMERIC ARG
T
```

and (ERROR 'FOO "NOT A FUNCTION") prints FOO NOT A FUNCTION. If both $MESS_1$ and $MESS_2$ are NIL, the message printed is simply ERROR.

If *NOBREAK* = T, ERROR prints its message and then calls ERROR! (below). Otherwise it calls (ERRORX '(17 ($MESS_1$. $MESS_2$))), i.e., generates error number 17, in which case the decision as to whether to break, and whether to print a message, is handled as any other error.

If the value of HELPFLAG (see the Controlling When to Break section above) is BREAK!, a break will always occur, irregardless of the value of *NOBREAK*.

If ERROR causes a break, the "break expression" is (ERROR $MESS_1$ $MESS_2$ NOBREAK). Using the GO**,** OK, , or EVAL break commands (see the Break Commands section above) will simply call ERROR again. It is sometimes helpful to design programs that call ERROR such that if the call to ERROR returns (as the result of using the RETURN break command), the operation is tried again. This lrts you fix any problems within the break environment, and try to continue the operation.

(**HELP** $MESS_1$ $MESS_2$ *BRKTYPE*) [Function]

Prints $MESS_1$ and $MESS_2$ similar to ERROR, and then calls BREAK1 passing *BRKTYPE* as the BRKTYPE argument. If both $MESS_1$ and $MESS_2$ are NIL, Help! is used for the message. HELP is a convenient way to program a default condition, or to terminate some portion of a program which the computation is theoretically never supposed to reach.

(**SHOULDNT** *MESS*) [Function]

Useful in situations when a program detects a condition that should never occur. Calls `HELP` with the message arguments *MESS* and "`Shouldn't happen!`" and a `BRKTYPE` argument of `'ERRORX`.

(**ERROR!**) [Function]

Equivalent to `XCL:ABORT`, except that if no `ERRORSET` or `XCL:CATCH-ABORT` isa found, it unwinds to the top of the process.

(**RESET**) [Function]

Programmable Control-D; immediately returns to the top level.

LAST-CONDITION [Variable]

Value is the condition object most recently signaled.

(**SETERRORN** *NUM MESS*) [Function]

Converts its arguments into a condition, then sets the value of `*LAST-CONDITION*` to the result.

(**ERRORMESS** *U*) [Function]

Prints message corresponding to its first argument. For example, `(ERRORMESS '(17 T))` would print: `T is not a LIST`

(**ERRORMESS1** $MESS_1$ $MESS_2$ $MESS_3$) [Function]

Prints the message corresponding to a `HELP` or `ERROR` break.

(**ERRORSTRING** *X*) [Function]

Returns as a new string the message corresponding to error number *X*, e.g., `(ERRORSTRING 10) = "NON-NUMERIC ARG"`.

Catching Errors

All error conditions are not caused by program bugs. For some programs, it is reasonable for some errors to occur (such as file not found errors) and it is possible for the program to handle the error itself. There are a number of functions that allow a program to "catch" errors, rather than abort the computation or cause a break.

(**ERRORSET** *FORM FLAG*) [Function]

Performs `(EVAL` *FORM*`)`. If no error occurs in the evaluation of *FORM*, the value of `ERRORSET` is a list containing one element, the value of `(EVAL` *FORM*`)`. If an error did occur, the value of `ERRORSET` is `NIL`.

`ERRORSET` is a lambda function, so its arguments are evaluated *before* it is entered, i.e., `(ERRORSET X)` means `EVAL` is called with the *value* of `X`. In most cases, `ERSETQ` and `NLSETQ` (below) are more useful.

Note: Beginning with the Medley release, there are no longer frames named ERRORSET on the stack and any programs that explicity look for them must be changed.

Performance Note: When a call to ERSETQ or NLSETQ is compiled, the form to be evaluated is compiled as a separate function. However, compiling a call to ERRORSET does not compile *FORM*. Therefore, if *FORM* performs a lengthy computation, using ERSETQ or NLSETQ can be much more efficient than using ERRORSET.

The argument *FLAG* controls the printing of error messages if an error occurs. If a *break* occurs below an ERRORSET, the message is printed regardless of the value of *FLAG*.

If *FLAG* = T, the error message is printed; if *FLAG* = NIL, the error message is not printed (unless NLSETQGAG is NIL, see below).

If *FLAG* = INTERNAL, this ERRORSET is ignored for the purpose of deciding whether or not to break or print a message (see the Controlling When to Break section above). However, the ERRORSET is in effect for the purpose of flow of control, i.e., if an error occurs, this ERRORSET returns NIL.

If *FLAG* = NOBREAK, no break will occur, even if the time criterion for breaking is met (the Controlling When to Break section above). *FLAG* = NOBREAK will *not* prevent a break from occurring if the error occurs more than HELPDEPTH function calls below the errorset, since BREAKCHECK will stop searching before it reaches the ERRORSET. To guarantee that no break occurs, you would also either have to reset HELPDEPTH or HELPFLAG.

(**ERSETQ** *FORM*) [NLambda Function]

Evaluates *FORM*, letting a break happen if an error occurs, but 9^ brings you back to the ERSETQ. Performs (ERRORSET '*FORM* T), printing error messages.

(**NLSETQ** *FORM*) [NLambda Function]

Evaluates *FORM*, witout breaking, returning NIL if an error occurs or a list containing *FORM* if no error occurs. Performs (ERRORSET '*FORM* NIL), without printing error messages.

NLSETQGAG [Variable]

If NLSETQGAG is NIL, error messages will print, regardless of the *FLAG* argument of ERRORSET. NLSETQGAG effectively changes all NLSETQs to ERSETQs. NLSETQGAG is initially T.

Changing and Restoring System State

In Medley, a computation can be interrupted / aborted at any point due to an error, or more forcefully, because a Control-D was typed, causing return to the top level. This situation creates problems for programs that need to perform a computation with the system in a “different state”, e.g., different radix, input file, readtable, etc. but want to be able to restore the state when the computation has completed. While program errors and Control-E are “caught” by ERRORSETs, Control-D is not. The program could redefine Control-D as a user interrupt (see Chapter 30), check for it, reenable it, and call RESET or something similar. Thus the system may be left in its changed state as a result of the computation being aborted. The following functions address this problem.

These functions cannot handle the situation where their environment is exited via anything other than a normal return, an error, or a reset. Therefore, a RETEVAL, RETFROM, RESUME, etc., will never be seen.

(**RESETLST** *FORM*$_1$ . . . *FORM*$_N$) [NLambda NoSpread Function]

RESETLST evaluates its arguments in order, after setting up an ERRORSET so that any reset operations performed by RESETSAVE (see below) are restored when the forms have been evaluated (or an error occurs, or a Control-D is typed). If no error occurs, the value of RESETLST is the value of *FORM*$_N$, otherwise RESETLST generates an error (after performing the necessary restorations).

RESETLST compiles open.

(**RESETSAVE** *X Y*) [NLambda NoSpread Function]

RESETSAVE is used within a call to RESETLST to change the system state by calling a function or setting a variable, while specifying how to restore the original system state when the RESETLST is exited (normally, or with an error or Control-D).

If *X* is atomic, resets the top level value of *X* to the value of *Y*. For example, (RESETSAVE LISPXHISTORY EDITHISTORY) resets the value of LISPXHISTORY to the value of EDITHISTORY, and provides for the original value of LISPXHISTORY to be restored when the RESETLST completes operation, (or an error occurs, or a Control-D is typed).

Note: If the variable is simply rebound, the RESETSAVE will not affect the most recent binding but will change only the top level value, and therefore probably not have the intended effect.

If *X* is not atomic, it is a form that is evaluated. If *Y* is NIL, *X* must return as its value its "former state", so that the effect of evaluating the form can be reversed, and the system state can be restored, by applying CAR of *X* to the value of *X*. For example, (RESETSAVE (RADIX 8)) performs (RADIX 8), and provides for RADIX to be reset to its original value when the RESETLST completes by applying RADIX to the value returned by (RADIX 8).

In the special case that CAR of *X* is SETQ, the SETQ is transparent for the purposes of RESETSAVE, i.e. you could also have written (RESETSAVE (SETQ X (RADIX 8))), and restoration would be performed by applying RADIX, not SETQ, to the previous value of RADIX.

If *Y* is not NIL, it is evaluated (before *X*), and its *value* is used as the restoring expression. This is useful for functions which do not return their "previous setting". For example,

```
[RESETSAVE (SETBRK ...) (LIST 'SETBRK (GETBRK]
```

will restore the break characters by applying SETBRK to the value returned by (GETBRK), which was computed before the (SETBRK ...) expression was evaluated. The restoration expression is "evaluated" by *applying* its CAR to its CDR. This insures that the "arguments" in the CDR are not evaluated again.

If *X* is `NIL`, *Y* is still treated as a restoration expression. Therefore,

```
(RESETSAVE NIL (LIST 'CLOSEF FILE))
```

will cause `FILE` to be closed when the `RESETLST` that the `RESETSAVE` is under completes (or an error occurs or a Control-D is typed).

`RESETSAVE` can be called when *not* under a `RESETLST`. In this case, the restoration is performed at the next `RESET`, i.e., Control-D or call to `RESET`. In other words, there is an "implicit" `RESETLST` at the top-level executive.

`RESETSAVE` compiles open. Its value is not a "useful" quantity.

(**RESETVAR** *VAR NEWVALUE FORM*) [NLambda Function]

Simplified form of `RESETLST` and `RESETSAVE` for resetting and restoring global variables. Equivalent to `(RESETLST (RESETSAVE` *VAR NEWVALUE*`)` *FORM*`)`. For example, `(RESETVAR LISPXHISTORY EDITHISTORY (FOO))` resets `LISPXHISTORY` to the value of `EDITHISTORY` while evaluating `(FOO)`. `RESETVAR` compiles open. If no error occurs, its value is the value of *FORM*.

(**RESETVARS** *VARSLST* E_1 E_2 ... E_N) [NLambda NoSpread Function]

Similar to `PROG`, except that the variables in *VARSLST* are global variables. In a deep bound system (like Medley), each variable is "rebound" using `RESETSAVE`.

In a shallow bound system (like Interlisp-10) `RESETVARS` and `PROG` are identical, except that the compiler insures that variables bound in a `RESETVARS` are declared as `SPECVARS` (see Chapter 18).

`RESETVARS`, like `GETATOMVAL` and `SETATOMVAL` (see Chapter 2), is provided to permit compatibility (i.e. transportablility) between a shallow bound and deep bound system with respect to conceptually global variables.

Note: Like `PROG`, `RESETVARS` returns `NIL` unless a `RETURN` statement is executed.

(**RESETFORM** *RESETFORM* $FORM_1$ $FORM_2$ ... $FORM_N$) [NLambda NoSpread Function]

Simplified form of `RESETLST` and `RESETSAVE` for resetting a system state when the corresponding function returns as its value the "previous setting." Equivalent to `(RESETLST (RESETSAVE` *RESETFORM*`)` $FORM_1$ $FORM_2$ ... $FORM_N$`)`. For example, `(RESETFORM (RADIX 8) (FOO))`. `RESETFORM` compiles open. If no error occurs, it returns the value returned by $FORM_N$.

For some applications, the restoration operation must be different depending on whether the computation completed successfully or was aborted somehow (e.g., by an error or by typing Control-D). To facilitate this, while the restoration operation is being performed, the value of `RESETSTATE` is bound to `NIL`, `ERROR`, `RESET`, or `HARDRESET` depending on whether the exit was normal, due to an error, due to a reset (i.e., Control-D), or due to call to `HARDRESET` (see Chapter 23). As an example of the use of `RESETSTATE`,

```
(RESETLST
  (RESETSAVE (INFILE X)
```

```
(LIST '[LAMBDA (FL)
       (COND ((EQ RESETSTATE 'RESET)
              (CLOSEF FL)
               DELFILE FL]
                     X))
                     FORMS)
```

will cause X to be closed and deleted only if a Control-D was typed during the execution of *FORMS*.

When specifying complicated restoring expressions, it is often necessary to use the old value of the saving expression. For example, the following expression will set the primary input file (to FL) and execute some forms, but reset the primary input file only if an error or Control-D occurs.

```
(RESETLST
          (SETQ TEM (INPUT FL))
          (RESETSAVE NIL
             (LIST '(LAMBDA (X) (AND RESETSTATE (INPUT X)))
                   TEM))
          FORMS)
```

So that you will not have to explicitely save the old value, the variable OLDVALUE is bound at the time the restoring operation is performed to the value of the saving expression. Using this, the previous example could be recoded as:

```
(RESETLST
          (RESETSAVE (INPUT FL)
             '(AND RESETSTATE (INPUT OLDVALUE)))
          FORMS)
```

As mentioned earlier, restoring is performed by applying CAR of the restoring expression to the CDR, so RESETSTATE and (INPUT OLDVALUE) will not be evaluated by the APPLY. This particular example works because AND is an nlambda function that explicitly evaluates its arguments, so APPLYing AND to (RESETSTATE (INPUT OLDVALUE)) is the same as EVALing (AND RESETSTATE (INPUT OLDVALUE)). PROGN also has this property, so you can use a lambda function as a restoring form by enclosing it within a PROGN.

The function RESETUNDO (see Chapter 13) can be used in conjunction with RESETLST and RESETSAVE to provide a way of specifying that the system be restored to its prior state by *undoing* the side effects of the computations performed under the RESETLST.

Error List

There are currently fifty-plus types of errors in Medley. Some of these errors are implementation dependent, i.e., appear in Medley but may not appear in other Interlisp systems. The error number is set internally by the code that detects the error before it calls the error handling functions, and is used by ERRORMESS for printing error messages.

Most errors will print the offending expression as part of the error message. Error number 18 (Control-B) always causes a break (unless HELPFLAG is NIL). All other errors cause breaks if BREAKCHECK returns T (see Controlling When to Break above).

The folloing error messages are arranged numerically with the printed message next to the error number. *X* is the offending expression in each error message. The obsolete error numbers still generate error messags, but they aren't particularly useful. For information on how to use the Common Lisp error conditions in your own programs, see *Common Lisp: the Language* by Steele.

0 Obsolete.

1 Obsolete.

2 **Stack Overflow**

Occurs when computation is too deep, either with respect to number of function calls, or number of variable bindings. Usually because of a non-terminating recursive computation, i.e., a bug. Condition type: STACK-OVERFLOW.

3 **RETURN to nonexistant block: *X***

Call to RETURN when not inside of an interpreted PROG. Condition type: ILLEGAL-RETURN.

4 ***X* is not a LIST**

RPLACA called on a non-list. Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'LIST

5 **Device error: *X***

An error with the local disk drive. Condition type: XCL:SIMPLE-DEVICE-ERROR *message*

6 **Serious condition XCL:ATTEMPT-TO-CHANGE-CONSTANT occured.**

Via SET or SETQ. Condition type: XCL:ATTEMPT-TO-CHANGE-CONSTANT

7 **Attempt to rplac NIL with *X***

Attempt either to RPLACA or to RPLACD NIL with something other than NIL. Condition type: XCL:ATTEMPT-TO-RPLAC-NIL *message*

8 **GO to a nonexistant tag: *X*.**

GO when not inside of a PROG, or GO to nonexistent label. Condition type: ILLEGAL-GO *tag*

9 **File won't open: *X***

From OPENSTREAM (see Chapter 24). Condition type: XCL:FILE-WONT-OPEN *pathname*

10 ***X* is not a NUMBER**

A numeric function e.g., PLUS, TIMES, GREATERP, expected a number and didn't get one. Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED TYPE 'CL:NUMBER

11 **Symbol name too long**

Attempted to create a symbol (via PACK, or typing one in, or reading from a file) with too many characters. In Medley, the maximum number of characters in a symbol is 255. Condition type: XCL:SYMBOL-NAME-TOO-LONG

12 **Symbol hash table full**

No room for any more (new) atoms. Condition type: XCL:SYMBOL-HT-FULL

13 **Stream not open: *X***

From an I/O function, e.g., READ, PRINT, CLOSEF. Condition type: XCL:STREAM-NOT-OPEN *stream*

14 ***X* is not a SYMBOL.**

SETQ, PUTPROP, GETTOPVAL, etc., given a non-atomic argument. Condition type: XCL:SMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'CL:SYMBOL

15 Obsolete

16 **`End of file X`**

From an input function, e.g., `READ`, `READC`, `RATOM`. After the error occurs, the file will still be left open. Condition type: `END-OF-FILE` *`stream`*

17 ***`X varying messages.`***

Call to `ERROR` (see Signalling Errors above). Condition type: `INTERLISP-ERROR MESSAGE`

18 Obsolete

19 **`Illegal stack arg: X`**

A stack function expected a stack position and was given something else. This might occur if the arguments to a stack function are reversed. Also occurs if you specified a stack position with a function name, and that function was not found on the stack (see Chapter 11). Condition type: `ILLEGAL-STACK-ARG` *`arg`*.

20 Obsolete

21 **`Array space full`**

System will first initiate a garbage collection of array space, and if no array space is reclaimed, will then generate this error. Condition type: `XCL:ARRAY-SPACE-FULL`.

22 **`File system resources exceeded`: *`X`***

Includes no more disk space, disk quota exceeded, directory full, etc. Condition type: `XCL:FS-RESOURCE-EXCEEDED`

23 **`File not found`**

File name does not correspond to a file in the corresponding directory. Can also occur if file name is ambiguous. Condition type: `XCL:FILE-NOT-FOUND` *pathname*

24 Obsolete

25 **`Invalid argument: X`**

A form ends in a non-list other than `NIL`, e.g., `(CONS T . 3)`. Condition type: `INVALID-ARGUMENT-LIST` *`argument`*

26 **`Hash table full`: *`X`***

See hash array functions, Chapter 6. Condition type: `XCL:HASH-TABLE-FULL` *`table`*

27 **`Invalid argument: X`**

Catch-all error. Currently used by `PUTD`, `EVALA`, `ARG`, `FUNARG`, etc. Condition type: `INVALID-ARGUMENT-LIST` *`argument`*

28 **`X is not a ARRAYP.`**

`ELT` or `SETA` given an argument that is not a legal array (see Chapter 5). Condition type: `XCL:SIMPLE-TYPE-ERROR` *`culprit`* `:EXPECTED-TYPE 'ARRAYP`

29 Obsolete

30 **`Stack ptr ahs been released NOBIND`**

A released stack pointer was supplied as a stack descriptor for a purpose other than as a stack pointer to be re-used (see Chapter 11). Condition type: `STACK-POINTER-REALEASED` *`name`*

31 **Serious condition XCL:STORAGE-EXHAUSTED occured.**

Following a garbage collection, if not enough words have been collected, and there is no unallocated space left in the system, this error is generated. Condition type: XCL:STORAGE-EXHAUSTED

32 Obsolete

33 Obsolete

34 **No more data types available**

All available user data types have been allocated (see Chapter 8). Condition type: XCL:DATA-TYPES-EXHAUSTED

35 **Serious condition XCL:ATTEMPT-TO-CHANGE-CONSTANT occured.**

In a PROG or LAMBDA expression. Condition type: XCL:ATTEMPT-TO-CHANGE-CONSTANT

36 Obsolete

37 Obsolete

38 ***X* is not a READTABLEP.**

The argument was expected to be a valid read table (see Chapter 25). Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'READTABLEP

39 ***X* is not a TERMTABLEP.**

The argument was expected to be a valid terminal table (see Chapter 30). Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'TERMTABLEP

40 Obsolete

41 **Protection violation**: ***X***

Attempt to open a file that you do not have access to. Also reference to unassigned device. Condition type: XCL:FS-PROTECTION-VIOLATION

42 **Invalid pathname: *X***

Illegal character in file specification, illegal syntax, e.g. two ;'s etc. Condition type: XCL:INVALID-PATHNAME *pathname*

43 Obsolete

44 ***X* is an unbound variable**

This occurs when a variable (symbol) was used which had neither a stack binding (wasn't an argument to a function nor a PROG variable) nor a top level value. The "culprit" ((CADR ERRORMESS)) is the symbol. If DWIM corrects the error, no error occurs and the error number is not set. However, if an error is going to occur, whether or not it will cause a break, the error number will be set. Condition type: UNBOUND-VARIABLE *name*

45 **Serious condition UNDEFINED-CAR-OF-FORM occured.**

Undefined function error. This occurs when a form is evaluated whose function position (CAR) does not have a definition as a function. Condition type: UNDEFINE-CAR-OF FORM *function*

46 ***X varying messages.***

This error is generated if `APPLY` is given an undefined function. Culprit is *`(LIST FN ARGS)`* Condition type: `UNDEFINED-FUNCTION-IN-APPLY`

47 **CONTROL E**

Control-E was typed. Condition type: `XCL:CONTROL-E-INTERRUPT`

48 **Floating point underflow.**

Underflow during floating-point operation. Condition type: `XCL:FLOATING-UNDERFLOW`

49 **Floating point overflow.**

Overflow during floating-point operation. Condition type: `XCL:OVERFLOW`

50 Obsolete

51 ***X* is not a HASH-TABLE**

Hash array operations given an argument that is not a hash array. Condition type: `XCL:SIMPLE-TYPE-ERROR` *`culprit`* `:EXPECTED-TYPE 'CL:HASH-TABLE`

52 **Too many arguments to *X***

Too many arguments given to a lambda-spread, lambda-nospread, or nlambda-spread function.

Medley does not cause an error if more arguments are passed to a function than it is defined with. This argument occurs when more individual arguments are passed to a function than Medley can store on the stack at once. The limit is currently 80 arguments.

In addition, many system functions, e.g., `DEFINE`, `ARGLIST`, `ADVISE`, `LOG`, `EXPT`, etc, also generate errors with appropriate messages by calling `ERROR` (see Signalling Errors above) which causes error number 17. Condition type: `TOO-MANY-ARGUMENTS` *`callee`* `:MAXIMUM CL:CALL-ARGUMENTS-LIMIT`

[This page intentionally left blank]

15. BREAKING, TRACING, AND ADVISING

Medley provides several different facilities for modifing the behavior of a function without actually editing its definition. By “breaking” a function, you can cause breaks to occur at various times in the running of an incomplete program, so that the program state can be inspected. “Tracing” a function causes information to be printed every time the function is entered or exited.

“Advising” is a facility for specifying longer-term function modifications. Even system functions can be changed through advising.

Breaking Functions and Debugging

Debugging a collection of Lisp functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. In the Medley, there are three facilities which allow you to (temporarily) modify selected function definitions so that you can follow the flow of control in your programs, and obtain this debugging information. All three redefine functions in terms of a system function, BREAK1 (see Chapter 14).

BREAK modifies the definition of a function *FN*, so that whenever *FN* is called and a break condition (user-defined) is satisfied, a function break occurs. You can then interrogate the state of the machine, perform any computation, and continue or return from the call.

TRACE modifies a definition of a function *FN* so that whenever *FN* is called, its arguments (or some other user-specified values) are printed. When the value of *FN* is computed it is printed also. TRACE is a special case of BREAK.

BREAKIN allows you to insert a breakpoint inside an expression defining a function. When the breakpoint is reached and if a break condition (defined by you) is satisfied, a temporary halt occurs and you can again investigate the state of the computation.

The following two examples illustrate these facilities. In the first example, the function FACTORIAL is traced. TRACE redefines FACTORIAL so that it print its arguments and value, and then goes on with the computation. When an error occurs on the fifth recursion, a full interactive break occurs. The situation is then the same as though (BREAK FACTORIAL) had been performed instead of (TRACE FACTORIAL), now you can evaluate various Interlisp forms and direct the course of the computation. In this case, the variable N is examined, and BREAK1 is instructed to return 1 as the value of this cell to FACTORIAL. The rest of the tracing proceeds without incident. Presumably, FACTORIAL would be edited to change L to 1.

```
←PP FACTORIAL
(FACTORIAL
        [LAMBDA (N)
        (COND
           ((ZEROP N)
              L)
           (T (ITIMES N (FACTORIAL (SUB1 N])
FACTORIAL
←(TRACE FACTORIAL)
(FACTORIAL)
←(FACTORIAL 4)
FACTORIAL:
N = 4
```

```
    FACTORIAL:
    N = 3
        FACTORIAL:
        N = 2
            FACTORIAL:
            N = 1
                FACTORIAL:
                N = 0
UNBOUND ATOM
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
                FACTORIAL = 1
            FACTORIAL = 1
        FACTORIAL = 2
    FACTORIAL = 6
FACTORIAL = 24
24
←
```

In the second example, a non-recursive definition of FACTORIAL has been constructed. BREAKIN is used to insert a call to BREAK1 just after the PROG label LOOP. This break is to occur only on the last two iterations, when N is less than 2. When the break occurs, in trying to look at the value of N, NN is mistakenly typed. The break is maintained, however, and no damage is done. After examining N and M the computation is allowed to continue by typing OK. A second break occurs after the next iteration, this time with N = 0. When this break is released, the function FACTORIAL returns its value of 120.

```
←PP FACTORIAL
(FACTORIAL
  [LAMBDA (N)
     (PROG ((M 1))
        LOOP (COND
                   ((ZEROP N)
                    (RETURN M)))
               (SETQ M (ITIMES M N))
               (SETQ N (SUB1 N))
               (GO LOOP])
FACTORIAL

←(BREAKIN FACTORIAL (AFTER LOOP) (ILESSP N 2]
SEARCHING...
FACTORIAL

←((FACTORIAL 5)
((FACTORIAL) BROKEN)
:NN
U.B.A.
NN
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)
```

```
((FACTORIAL) BROKEN)
:N
0
:OK
(FACTORIAL)
120
←
```

Note: BREAK and TRACE can also be used on CLISP words which appear as CAR of form, e.g. FETCH, REPLACE, IF, FOR, DO, etc., even though these are not implemented as functions. For conditional breaking, you can refer to the entire expression via the variable EXP, e.g. (BREAK (FOR (MEMB 'UNTIL EXP))).

(**BREAK0** *FN WHEN COMS* — —) [Function]

Sets up a break on the function *FN*; returns *FN*. If *FN* is not defined, returns (*FN* NOT DEFINED).

The value of *WHEN*, if non-NIL, should be an expression that is evaluated whenever *FN* is entered. If the value of the expression is non-NIL, a break is entered, otherwise the function simply called and returns without causing a break. This provides the means of conditionally breaking a function.

The value of *COMS*, if non-NIL, should be a list of break commands, that are interpreted and executed if a break occurs. (See the *BRKCOMS* argument to BREAK1, Chapter 14.)

BREAK0 sets up a break by doing the following:

Redefines *FN* as a call to BREAK1 (Chapter 14), passing an equivalent definition of *FN*, *WHEN*, *FN*, and *COMS* as the BRKEXP, *BRKWHEN*, *BRKFN*, and *BRKCOMS* arguments to BREAK1

Defines a GENSYM (Chapter 2) with the original definition of *FN*, and puts it on the property list of *FN* under the property BROKEN

Puts the form (BREAK0 *WHEN COMS*) on the property list of *FN* under the property BRKINFO (for use in conjunction with REBREAK)

Adds *FN* to the front of the list BROKENFNS.

If *FN* is non-atomic and of the form (FN_1 IN FN_2), BREAK0 breaks every call to FN_1 from within FN_2. This is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g., (RPLACA IN FOO), (PRINT IN FIE), etc. It is similar to BREAKIN described below, but can be performed even when FN_2 is compiled or blockcompiled, whereas BREAKIN only works on interpreted functions. If FN_1 is not found in FN_2, BREAK0 returns the value (FN_1 NOT FOUND IN FN_2).

BREAK0 breaks one function *inside* another by first calling a function which changes the name of FN_1 wherever it appears inside of FN_2 to that of a new function, *FN1*-IN-*FN2*, which is initially given the same function definition as FN_1. Then BREAK0 proceeds to break on *FN1*-IN-*FN2* exactly as described above. In addition to breaking *FN1*-IN-*FN2* and adding *FN1*-IN-*FN2* to the list BROKENFNS, BREAK0 adds *FN1* to the property value for the property NAMESCHANGED on the property list of FN_2 and puts (FN_2 . FN_1) on the property list of *FN1*-IN-*FN2* under the property ALIAS. This will enable UNBREAK to recognize what changes have been made and restore the function FN_2 to its original state.

If *FN* is nonatomic and not of the above form, BREAK0 is called for each member of *FN* using the same values for *WHEN*, *COMS*, and *FILE*. This distributivity permits you to specify complicated break conditions on several functions. For example,

```
(BREAK0 '(FOO1 ((PRINT PRIN1) IN (FOO2 FOO3)))
   '(NEQ X T)
   '(EVAL ?= (Y Z) OK) )
```

will break on FOO1, PRINT-IN-FOO2, PRINT-IN-FOO3, PRIN1-IN-FOO2 and PRIN1-IN-FOO3.

If *FN* is non-atomic, the value of BREAK0 is a list of the functions broken.

(**BREAK** *X*) [NLambda NoSpread Function]

For each atomic argument, it performs (BREAK0 *ATOM* T). For each list, it performs (APPLY 'BREAK0 *LIST*). For example, (BREAK FOO1 (FOO2 (GREATERP N 5) (EVAL))) is equivalent to (BREAK0 'FOO1 T) and (BREAK0 'FOO2 '(GREATERP N 5) '(EVAL)).

(**TRACE** *X*) [NLambda NoSpread Function]

For each atomic argument, it performs (BREAK0 *ATOM* T '(TRACE ?= NIL GO)). The flag TRACE is checked for in BREAK1 and causes the message "*FUNCTION* :" to be printed instead of (*FUNCTION* BROKEN).

For each list argument, CAR is the function to be traced, and CDR the forms to be viewed, i.e., TRACE performs:

```
(BREAK0 (CAR LIST) T (LIST 'TRACE '?= (CDR LIST) 'GO))
```

For example, (TRACE FOO1 (FOO2 Y)) causes both FOO1 and FOO2 to be traced. All the arguments of FOO1 are printed; only the value of Y is printed for FOO2. In the special case when you want to see *only* the value, you can perform (TRACE (*FUNCTION*)). This sets up a break with commands (TRACE ?= (NIL) GO).

Note: You can always call BREAK0 to obtain combination of options of BREAK1 not directly available with BREAK and TRACE. These two functions merely provide convenient ways of calling BREAK0, and will serve for most uses.

Note: BREAK0, BREAK, and TRACE print a warning if you try to modify a function on the list UNSAFE.TO.MODIFY.FNS (Chapter 10).

(**BREAKIN** *FN WHERE WHEN COMS*) [NLambda Function]

BREAKIN enables you to insert a break, i.e., a call to BREAK1 (Chapter 14), at a specified location in the interpreted function *FN*. BREAKIN can be used to insert breaks before or after PROG labels, particular SETQ expressions, or even the evaluation of a variable. This is because BREAKIN operates by calling the editor and actually inserting a call to BREAK1 at a specified point *inside* of the function. If *FN* is a compiled function, BREAKIN returns (*FN* UNBREAKABLE) as its value.

WHEN should be an expression that is evaluated whenever the break is entered. If the value of the expression is non-NIL, a break is entered, otherwise the function simply called and returns without causing a break. This provides the means of creating a conditional break. For BREAKIN, unlike BREAK0, if *WHEN* is NIL, it defaults to T.

COMS, if non-NIL, should be a list of break commands, that are interpreted and executed if a break occurs. (See the *BRKCONMS* argument to BREAK1, Chapter 14.)

WHERE specifies where in the definition of *FN* the call to BREAK1 is to be inserted. *WHERE* should be a list of the form (BEFORE ...), (AFTER ...), or (AROUND ...). You specify where the break is to be inserted by a sequence of editor commands, preceded by one of the symbols BEFORE, AFTER, or AROUND, which BREAKIN uses to determine what to do once the editor has found the specified point, i.e., put the call to BREAK1 BEFORE that point, AFTER that point, or AROUND that point. For example, (BEFORE COND) will insert a break before the first occurrence of COND, (AFTER COND 2 1) will insert a break after the predicate in the first COND clause, (AFTER BF (SETQ X &)) after the *last* place X is set. Note that (BEFORE TTY:) or (AFTER TTY:) permit you to type in commands to the editor, locate the correct point, and verify it, and exit from the editor with OK. BREAKIN then inserts the break BEFORE, AFTER, or AROUND that point.

Note: A STOP command typed to TTY: produces the same effect as an unsuccessful edit command in the original specification, e.g., (BEFORE CONDD). In both cases, the editor aborts, and BREAKIN types (NOT FOUND).

If *WHERE* is (BEFORE ...) or (AFTER ...), the break expression is NIL, since the value of the break is irrelevant. For (AROUND ...), the break expression will be the indicated form. In this case, you can use the EVAL command to evaluate that form, and examine its value, before allowing the computation to proceed. For example, if you inserted a break after a COND predicate, e.g., (AFTER (EQUAL X Y)), you would be powerless to alter the flow of computation if the predicate were not true, since the break would not be reached. However, by breaking (AROUND (EQUAL X Y)), you can evaluate the break expression, i.e., (EQUAL X Y), look at its value, and return something else if desired.

If *FN* is interpreted, BREAKIN types SEARCHING... while it calls the editor. If the location specified by *WHERE* is not found, BREAKIN types (NOT FOUND) and exits. If it is found, BREAKIN puts T under the property BROKEN-IN and (*WHERE WHEN COMS*) under the the property BRKINFO on the property list of *FN*, and adds *FN* to the front of the list BROKENFNS.

Multiple break points, can be inserted with a single call to BREAKIN by using a list of the form ((BEFORE ...) ... (AROUND ...)) for *WHERE*. It is also possible to call BREAK or TRACE on a function which has been modified by BREAKIN, and conversely to BREAKIN a function which has been redefined by a call to BREAK or TRACE.

The message typed for a BREAKIN break is ((*FN*) BROKEN), where *FN* is the name of the function inside of which the break was inserted. Any error, or typing control-E, will cause the full identifying message to be printed, e.g., (FOO BROKEN AFTER COND 2 1).

A special check is made to avoid inserting a break inside of an expression headed by any member of the list NOBREAKS, initialized to (GO QUOTE *), since this break would never be activated. For example, if (GO L) appears before the label L, BREAKIN (AFTER L) will not insert the break inside of the GO expression, but skip this occurrence of L and go on to the next L, in this case the label L. Similarly, for BEFORE or AFTER breaks, BREAKIN checks to make sure that the break is being inserted at a “safe” place. For example, if you request a break (AFTER X) in (PROG ... (SETQ X &) ...), the break will actually be inserted after (SETQ X &), and a message printed to this effect, e.g., BREAK INSERTED AFTER (SETQ X &).

(**UNBREAK** *X*) [NLambda NoSpread Function]

UNBREAK takes an indefinite number of functions modified by BREAK, TRACE, or BREAKIN and restores them to their original state by calling UNBREAK0. Returns list of values of UNBREAK0.

(UNBREAK) will unbreak all functions on BROKENFNS, in reverse order. It first sets BRKINFOLST to NIL.

(UNBREAK T) unbreaks just the first function on BROKENFNS, i.e., the most recently broken function.

(**UNBREAK0** *FN* —) [Function]

Restores *FN* to its original state. If *FN* was not broken, value is (NOT BROKEN) and no changes are made. If *FN* was modified by BREAKIN, UNBREAKIN is called to edit it back to its original state. If *FN* was created from (FN_1 IN FN_2), (i.e., if it has a property ALIAS), the function in which *FN* appears is restored to its original state. All dummy functions that were created by the break are eliminated. Adds property value of BRKINFO to the front of BRKINFOLST.

Note: (UNBREAK0 '(FN_1 IN FN_2)) is allowed: UNBREAK0 will operate on (FN_1-IN-FN_2) instead.

(**UNBREAKIN** *FN*) [Function]

Performs the appropriate editing operations to eliminate all changes made by BREAKIN. *FN* may be either the name or definition of a function. Value is *FN*.

UNBREAKIN is automatically called by UNBREAK if *FN* has property BROKEN-IN with value T on its property list.

(**REBREAK** *X*) [NLambda NoSpread Function]

Nlambda nospread function for rebreaking functions that were previously broken without having to respecify the break information. For each function on *X*, REBREAK searches BRKINFOLST for break(s) and performs the corresponding operation. Value is a list of values corresponding to calls to BREAK0 or BREAKIN. If no information is found for a particular function, returns (*FN* - NO BREAK INFORMATION SAVED).

(REBREAK) rebreaks everything on BRKINFOLST, so (REBREAK) is the inverse of (UNBREAK).

(REBREAK T) rebreaks just the first break on BRKINFOLST, i.e., the function most recently unbroken.

(**CHANGENAME** *FN FROM TO*) [Function]

Replaces all occurrences of *FROM* by *TO* in the definition of *FN*. If *FN* is defined by an expr definition, CHANGENAME performs (ESUBST *TO FROM* (GETD *FN*)) (see Chapter 16). If *FN* is compiled, CHANGENAME searches the literals of *FN* (and all of its compiler generated subfunctions), replacing each occurrence of *FROM* with *TO*.

Note that *FROM* and *TO* do not have to be functions, e.g., they can be names of variables, or any other literals.

CHANGENAME returns *FN* if at least one instance of *FROM* was found, otherwise NIL.

(**VIRGINFN** *FN FLG*) [Function]

The function that knows how to restore functions to their original state regardless of any amount of breaks, breakins, advising, compiling and saving exprs, etc. It is used by PRETTYPRINT, DEFINE, and the compiler.

If *FLG* = NIL, as for PRETTYPRINT, it does not modify the definition of *FN* in the process of producing a “clean” version of the definition; it works on a copy.

If *FLG* = T, as for the compiler and DEFINE, it physically restores the function to its original state, and prints the changes it is making, e.g., FOO UNBROKEN, FOO UNADVISED, FOO NAMES RESTORED, etc.

Returns the virgin function definition.

Advising

The operation of advising gives you a way of modifying a function without necessarily knowing how the function works or even what it does. Advising consists of modifying the *interface* between functions as opposed to modifying the function definition itself, as in editing. BREAK, TRACE, and BREAKDOWN, are examples of the use of this technique: they each modify user functions by placing relevant computations *between* the function and the rest of the programming environment.

The principal advantage of advising, aside from its convenience, is that it allows you to treat anyone’s functions as “black boxes,” and to modify them without concern for their contents or details of operations. For example, you could modify SYSOUT to set SYSDATE to the time and date of creation by (ADVISE 'SYSOUT '(SETQ SYSDATE (DATE))).

As with BREAK, advising works equally well on compiled and interpreted functions. Similarly, it is possible to make a change which only operates when a function is called from some other specified function. For example, you can modify the interface between two particular functions, instead of the interface between one function and the rest of the world. This latter feature is especially useful for changing the *internal* workings of a system function.

For example, suppose you wanted TIME (Chapter 22) to print the results of your measurements to the file FOO instead of the terminal. You can accomplish this by (ADVISE '((PRIN1 PRINT SPACES) IN TIME) 'BEFORE '(SETQQ U FOO)).

Advising PRIN1, PRINT, or SPACES directly would have affected all calls to these frequently used functions, whereas advising ((PRIN1 PRINT SPACES) IN TIME) affects just those calls to PRIN1, PRINT, and SPACES from TIME.

Advice can also be specified to operate after a function has been evaluated. The value of the body of the original function can be obtained from the variable !VALUE, as with BREAK1.

Implementation of Advising

After a function has been modified several times by ADVISE, it will look like:

```
(LAMBDA arguments
   (PROG (!VALUE)
      (SETQ !VALUE
         (PROG NIL
              advice1
                 .
                 .        advice before
                 .
              advicen
              (RETURN BODY)))
        advice1
           .
           .        advice after
           .
        advicem
        (RETURN !VALUE)))
```

where *BODY* is equivalent to the original definition. If *FN* was originally an expr definition, *BODY* is the body of the definition, otherwise a form using a GENSYM which is defined with the original definition.

The structure of a function modified by ADVISE allows a piece of advice to bypass the original definition by using the function RETURN. For example, if (COND ((ATOM X**)** (RETURN Y))) were one of the pieces of advice *before* a function, and this function was entered with X atomic, Y would be returned as the value of the inner PROG, !VALUE would be set to Y, and control passed to the advice, if any, to be executed AFTER the function. If this same piece of advice appeared *after* the function, Y would be returned as the value of the entire advised function.

The advice (COND ((ATOM X) (SETQ !VALUE Y))) *after* the function would have a similar effect, but the rest of the advice *after* the function would still be executed.

Note: Actually, ADVISE uses its own versions of PROG, SETQ, and RETURN, (called ADV-PROG, ADV-SETQ, and ADV-RETURN) to enable advising these functions.

Advise Functions

ADVISE is a function of four arguments: *FN*, *WHEN*, *WHERE*, and *WHAT*. *FN* is the function to be modified by advising, *WHAT* is the modification, or piece of advice. *WHEN* is either BEFORE, AFTER, or AROUND, and indicates whether the advice is to operate BEFORE, AFTER, or AROUND the body of the function definition. *WHERE* specifies exactly where in the list of advice the new advice is to be placed, e.g., FIRST, or (BEFORE PRINT) meaning before the advice containing PRINT, or (AFTER 3) meaning after the third piece of advice, or even (: TTY:). If *WHERE* is specified, ADVISE first checks to see if it is one of LAST, BOTTOM, END, FIRST, or TOP, and operates accordingly. Otherwise, it constructs an appropriate edit command and calls the editor to insert the advice at the corresponding location.

Both *WHEN* and *WHERE* are optional arguments, in the sense that they can be omitted in the call to ADVISE. In other words, ADVISE can be thought of as a function of two arguments (ADVISE *FN WHAT*), or a function of three arguments: (ADVISE *FN WHEN WHAT*), or a function of four arguments: (ADVISE *FN WHEN WHERE WHAT*). Note that the advice is always the *last* argument. If *WHEN* = NIL, BEFORE is used. If *WHERE* = NIL, LAST is used.

(**ADVISE** *FN WHEN WHERE WHAT*) [Function]

FN is the function to be advised, *WHEN* = BEFORE, AFTER, or AROUND, *WHERE* specifies where in the advice list the advice is to be inserted, and *WHAT* is the piece of advice.

If *FN* is of the form (FN_1 IN FN_2), FN_1 is changed to *FN1*-IN-*FN2* throughout FN_2, as with break, and then *FN1*-IN-*FN2* is used in place of *FN*. If FN_1 and/or FN_2 are lists, they are distributed as with BREAK0.

If *FN* is broken, it is unbroken before advising.

If *FN* is not defined, an error is generated, NOT A FUNCTION.

If *FN* is being advised for the first time, i.e., if (GETP *FN* 'ADVISED) = NIL, a GENSYM is generated and stored on the property list of *FN* under the property ADVISED, and the GENSYM is defined with the original definition of *FN*. An appropriate expr definition is then created for *FN*, using private versions of PROG, SETQ, and RETURN, so that these functions can also be advised. Finally, *FN* is added to the (front of) ADVISEDFNS, so that (UNADVISE T) always unadvises the last function advised.

If *FN* has been advised before, it is moved to the front of ADVISEDFNS.

If *WHEN* = BEFORE or AFTER, the advice is inserted in *FN*'s definition either BEFORE or AFTER the original body of the function. Within that context, its position is determined by *WHERE*. If *WHERE* = LAST, BOTTOM, END, or NIL, the advice is added following all other advice, if any. If *WHERE* = FIRST or TOP, the advice is inserted as the first piece of advice. Otherwise, *WHERE* is treated as a command for the editor, similar to BREAKIN, e.g., (BEFORE 3), (AFTER PRINT).

If *WHEN* = AROUND, the body is substituted for * in the advice, and the result becomes the new body, e.g., (ADVISE 'FOO 'AROUND '(RESETFORM (OUTPUT T) *)). Note that if several pieces of AROUND advice are specified, earlier ones will be embedded inside later ones. The value of *WHERE* is ignored.

Finally (LIST *WHEN WHERE WHAT*) is added (by ADDPROP) to the value of property ADVICE on the property list of *FN*, so that a record of all the changes is available for subsequent use in readvising. Note that this property value is a list of the advice in order of calls to ADVISE, not necessarily in order of appearance of the advice in the definition of *FN*.

The value of ADVISE is *FN*.

If *FN* is non-atomic, every function in *FN* is advised with the same values (but copies) for *WHEN*, *WHERE*, and *WHAT*. In this case, ADVISE returns a list of individual functions.

Note: Advised functions can be broken. However if a function is broken at the time it is advised, it is first unbroken. Similarly, advised functions can be edited, including their advice. UNADVISE will still restore the function to its unadvised state, but any changes to the body of the definition will survive. Since the advice stored on the property list is the same structure as the advice inserted in the function, editing of advice can be performed on either the function's definition or its property list.

(**UNADVISE** *X*) [NLambda NoSpread Function]

An nlambda nospread like UNBREAK. It takes an indefinite number of functions and restores them to their original unadvised state, including removing the properties added by ADVISE. UNADVISE saves on the list ADVINFOLST enough information to allow restoring a function to its advised state using READVISE. ADVINFOLST and READVISE thus correspond to BRKINFOLST and REBREAK. If a function contains the property READVICE, UNADVISE moves the current value of the property ADVICE to READVICE.

(UNADVISE) unadvises all functions on ADVISEDFNS in reverse order, so that the most recently advised function is unadvised last. It first sets ADVINFOLST to NIL.

(UNADVISE T) unadvises the first function of ADVISEDFNS, i.e., the most recently advised function.

(**READVISE** *X*) [NLambda NoSpread Function]

An nlambda nospread like REBREAK for restoring a function to its advised state without having to specify all the advise information. For each function on *X*, READVISE retrieves the advise information either from the property READVICE for that function, or from ADVINFOLST, and performs the corresponding advise operation(s). It also stores this information on the property READVICE if not already there. If no information is found for a particular function, value is (*FN* - NO ADVICE SAVED).

(READVISE) readvises everything on ADVINFOLST.

(READVISE T) readvises the first function on ADVINFOLST, i.e., the function most recently unadvised.

A difference between ADVISE, UNADVISE, and READVISE versus BREAK, UNBREAK, and REBREAK, is that if a function is not rebroken between successive (UNBREAK)s, its break information is forgotten. However, once READVISE is called on a function, that function's advice is permanently saved on its property list (under READVICE); subsequent calls to

UNADVISE will not remove it. In fact, calls to UNADVISE update the property READVICE with the current value of the property ADVICE, so that the sequence READVISE, ADVISE, UNADVISE causes the augmented advice to become permanent. The sequence READVISE, ADVISE, READVISE removes the "intermediate advice" by restoring the function to its earlier state.

(**ADVISEDUMP** *X FLG*) [Function]

Used by PRETTYDEF when given a command of the form (ADVISE ...) or (ADVICE ...). If *FLG* = T, ADVISEDUMP writes both a DEFLIST and a READVISE; this corresponds to (ADVISE ...). If *FLG* = NIL, only the DEFLIST is written; this corresponds to (ADVICE ...). In either case, ADVISEDUMP copies the advise information to the property READVICE, thereby making it "permanent" as described above.

16. SEdit - The Structure Editor

Medley's code editors are "structure" editors—they know how to take advantage of Lisp code being represented as lists. One is a display editor named SEdit and the other is a TTY-based editor.

Starting the Editor

The editor is normally called using the following functions:

(`DF` *`FN`*) [NLambda NoSpread Function]

Edit the definition of the function *`FN`*. `DF` handles exceptional cases (the function is broken or advised, the definition is on the property list, the function needs to be loaded from a file, etc.) the same as `EDITF` (see below).

If you call `DF` with a name that has no function definition, you are prompted with a choise of definers to use.

(`DV` *`VAR`*) [NLambda NoSpread Function]

Edit the value of the variable *`VAR`*.

(`DP` *`NAME PROP`*) [NLambda NoSpread Function]

Edit property *`PROP`* of the symbol *`NAME`*. If *`PROP`* is not given, the whole property list of *`NAME`* is edited.

(`DC` *`FILE`*) [NLambda NoSpread Function]

Edit the file package commands (or "filecoms," see Chapter 17) for the file *`FILE`*.

(`ED` *`NAME OPTIONS`*) [Function]

This function finds out what kind of definiton *`NAME`* has and lets you edit it. If NAME has more than one definition (e.g., it's both a function and a macro), you will be prompted for the right one. If NAME has no definition, you'll be asked what kind of definition to create.

Choosing Your Editor

The default editor may be set with `EDITMODE`:

(`EDITMODE` *NEWMODE*) [Function]

If *NEWMODE* is `DISPLAY`, sets the default editor to be SEdit; or the teletype editor (if *NEWMODE* is `TELETYPE`). Returns the previous setting. If *NEWMODE* is `NIL`, returns the previous setting without setting a new editor.

SEdit - The Structure Editor

SEdit is a structure editor. You use a structure editor when you want to edit objects instead of text. SEdit is a part of the environment and operates directly on objects in the system you are running. SEdit behaves differently depending on the type of objects you are editing.

Common Lisp definitions: SEdit always edits a copy of a Common Lisp definition. The changes made while you edit a function will not be installed until the edit session is complete.

For example, when you edit a Common Lisp function, you edit the definition of the function and not the executable version of the function. When you end the session the comments will be stripped of the definition and the definition will be installed as the executable version of the function.

Interlisp functions and macros: SEdit edits the actual structure that will be run, except editing the source for a compiled function. In this case, changes are made and the function is unsaved when you complete the edit session.

All other structures: Variables, property lists and other structures are edited directly in place, i.e. SEdit installs all changes as they are made.

If you make a severe editing error, you can abort the edit session with an Abort command (see Command Keys, below). This command undoes all changes from the beginning of the edit session and exits from SEdit without changing your environment.

If you change the definition of an object that is being edited in an SEdit window, Medley will ask you if you want to throw away the changes made there.

SEdit supports the standard Copy-Select mechanism in Medley.

An SEdit Session

Whenever you call SEdit, a new SEdit window is created. This SEdit window has its own process. You can make edits in the window, shrink it while you do something else, expand it and edit some more, and finally close the window when you are done.

Throughout an edit session, SEdit remembers everything that you do IN a change history. You can undo and redo edits sequentially. When you end the edit session, SEdit forgets this information and installs the changes in the system.

You signal the end of the session in the following ways:

- Close the window.

- Shrink the window. If you expand the window again, you can continue editing.

- Issue a Completion Command, see below.

SEdit Carets

There are two carets in SEdit, the edit caret and the structure caret. The edit caret appears when characters are edited within a single symbol, string, or comment. Anything you type will appear at the edit caret as part of the item it's in. The edit caret looks like this:

```
(a b)
```

The structure caret appears when the edit point is between symbols (or strings or comments), so that anything you type will go into a new one. It looks like this:

```
(a b)
```

SEdit changes the caret frequently, depending on where the caret is positioned. The left mouse button positions the edit caret. The middle mouse button positions the structure caret.

The Mouse

The left mouse button selects parts of Lisp structures. The middle mouse button selects whole Lisp structures.

For example; select the `Q` in `LEQ` below by pressing the left mouse button when the pointer is over the `Q`.

```
(LEQ n 1)
```

Any characters you type in now will be appended to the symbol `LEQ`.

Selecting the same letter with the middle mouse button selects the whole symbol (this matches TEdit's character / word selection convention), and sets a structure caret between the `LEQ` and the `n`:

```
(LEQ n 1)
```

Any characters you type in now will form a new symbol between the `LEQ` and the `n`.

Larger structures can be selected in two ways. Use the middle mouse button to position the mouse cursor on the parenthesis of the list you want to edit. Press the mouse button multiple times, without moving the mouse, extends the selection. In the previous example, if the middle button was pressed twice, the list `(LEQ ...)` would be selected:

```
(LEQ n 1)
```

Press the button a third time and you will select the list containing the `(LEQ n 1)` to be selected.

The right mouse button positions the mouse cursor for selecting sequences of structures or substructures. Extended selections are indicated by a box enclosing the structures selected. The selection extends in the same mode as the original selection. That is, if the original selection was a character selection, the right button will be used to select more characters in the same atom. Extended selections also have the property of being marked for pending deletion. That is, the selection takes the place of the caret, and anything typed in is inserted in place of the selection.

For example, selecting the `E` by pressing the left mouse button and selecting the `Q` by pressing the right mouse button will produce:

```
(LEQ n 1)
```

Similarly, pressing the middle mouse button and then selecting with the right mouse button extends the selection by whole structures. In our example, pressing the middle mouse button to select `LEQ` and pressing the right mouse button to select the `1` will produce:

```
(LEQ n 1)
```

This is not the same as selecting the entire list, as above. Instead, the elements in the list are collectively selected, but the list itself is not.

Gaps

SEdit requires that everything edited must have an underlying Lisp structure at all times. Some characters, such as single quote “'” have no meaning by themselves, but must be followed by something more. Whenyou type such a character, SEdit puts a “gap” where the rest of the input should go. When you type, the gap is automatically replaced.

A gap looks like: -×-

After you type a quote, the gap looks like this: '$\boxed{-\times-}$ with the gap marked for pending deletion.

Broken Atoms

When you type an atom (a symbol or a number), SEdit saves the characters you type until you are finished. Typing any character that cannot belong to an atom, like a space or open parenthesis, ends the atom. SEdit then tries to create an atom with the characters you just typed, just as if they were read by the Lisp reader. The atom then becomes part of the structure you're editing.

If an error occurs when SEdit reads the atom, SEdit creates a structure called a Broken-Atom. A Broken-Atom looks and behaves just like a normal atom, but is printed in italics to tell you that something is wrong.

SEdit creates a Broken-Atom when the characters typed don't make a legal atom. For example, the characters "DECLARE:" can't be a symbol because the colon is a package specifier, but the form is not correct for a package-qualified symbol. Similarly, the characters "#b123" cannot represent an integer in base two, because 2 and 3 aren't legal digits in base two, so SEdit would make a Broken-Atom that looks like *#b123*.

You can edit Broken-Atoms just like real atoms. Whenever you finish editing a Broken-Atom, SEdit again tries to create an atom from the characters. If SEdit succeeds, it reprints the atom in SEdit's default font, rather than in italics. Be sure to correct any Broken-Atoms you create before exiting SEdit, since Broken-Atoms do not behave in any useful way outside SEdit.

Special Characters

Some characters have special meanings in Lisp, and are therefor treated specially by SEdit. SEdit must always have a complete structure to work on at any level of the edit. This means that SEdit needs a special way to type in structures such as lists, strings, and quoted objects. In most instances these structures can be typed in just as they would be to a regular Exec, but in the following cases this is not possible.

Lists: `( )` Lists begin with an open parenthesis character "(". Typing an open parenthesis gives a balanced list. SEdit inserts both an open and a close parenthesis. The structure caret is placed between the two parentheses. List elements can be typed in at the structure caret. When a close parenthesis, ")" is typed, the caret will be moved outside the list, effectively finishing the list. Square bracket characters, "[" and "]", have no special meaning in SEdit, as they have no special meaning in Common Lisp.

Single Quote: `'`
Backquote: `` ` ``
Comma: `,`
At Sign: `,@`
Dot: `,.`
Hash Quote: `#'` All these characters are special macro characters in Common Lisp. When you type one, SEdit will echo the character followed by a gap, which you should then fill in.

Dotted Lists: `( . )` Use period to enter dotted pairs. After you type a dot, SEdit prints a dot and a gap to fill in for the tail of the list. To dot an existing list, point the cursor between the last and second to last elements, and type a dot. To undot a list, select the tail of the list before the dot while holding down the SHIFT key.

Single escape: `\` or `%` Use the single escape characters to make symbols with special characters. The single escape character for Interlisp is "%". The single escape character for Common Lisp is "\".

When you want to create a symbol with a special character in it you have to type a single escape character before you type the character itself. SEdit does not echo the single escape character until you type the following character.

For example; create the Common Lisp symbol `APAREN-(`. Since SEdit normally will treat the "`(`" as the start of a new list you have to tell SEdit to treat it as an ordinary character. You do this by typing a "\" before you type the "`(`".

Multiple Escape: `|` Use the multiple escape character when you enter symbols with many special characters. SEdit always balances multiple escape characters. When you type one, SEdit adds another, with the caret between them. If you type a second vertical bar, the caret moves after it, but is still inthe same symbol, so you can add more unescaped characters.

Comment: **;** A semicolon starts a comment. When you type a semicolon, an empty comment is inserted with the caret in position to type the comment. Comments can be edited like strings.

There are three levels of comments supported by SEdit: single-, double-, and triple-semicolon. Single-semicolon comments are formatted at the comment column, about three-quarters of the way across the window. Double-semicolon comments are formatted at the current indentation of the code they are in. Triple semicolon comments are formatted against the left margin. The level of a comment can be increased or decreased by pointing after the semicolon, and either typing another semicolon, or backspacing over the preceding semicolon. Comments can be placed anywhere in your Common Lisp code. However, in Interlisp code, they must follow the placement rules for Interlisp comments.

String: **"** Enter strings in SEdit by typing a double quote. SEdit balances the double quotes: When one is typed, SEdit produces a second, with the caret between the two. If you type a double-quote in the middle of a string, SEdit breaks the string in two, leaving the caret between them.

SEdit Commands

Enter SEdit commands either from the keyboard or from the SEdit menu. When possible, SEdit uses a named key on the keyboard, e.g., the DELETE key. Other commands are combinations of Meta, Control, and alphabetic keys. For the alphabetic command keys, either uppercase or lowercase will work.

There are two menus available, as an alternative means of invoking commands. They are the middle button popup menu, and the attached command menu. These menus are described in more detail below.

Meta-A Abort the session. Throw away the changes made to the form.

Meta-B Change the Print Base. Prompts for entry of the desired Print Base, in decimal. SEdit redisplays fixed point numbers in this new base.

Control-C Tell SEdit that this session is complete and compiles the definition being edited. The variable `*COMPILE-FN*` determines which function to use as compiler. See the Options section below.

Control-Meta-C Signals the system that this edit is complete, compiles the definition being editing, and closes the window.

DELETE Deletes the current selection.

Meta-E Evaluate the current selection. If the result is a structure, the inspector is called on it, allowing the user to choose how to look at the result. Otherwise, the result is printed in the SEdit prompt window. The evaluation is done in the process from which the edit session was started. Thus, while editing a function from a break window, evaluations are done in the context of the break.

FIND
Meta-F Find a specified structure, or sequence of structures. If there is a current selection, SEdit looks for the next occurrence of the selected structure. If there is no selection, SEdit prompts for the structure to find, and searches forward from the position of the caret. The found structure will be selected, so the Find command can be used to easily find the same structure again.

If a sequence of structures are selected, SEdit will look for the next occurrence of the same sequence. Similarly, when SEdit prompts for the structure to find, you can type a sequence of structures to look for.

The variable `*WRAP-SEARCH*` controls whether or not SEdit wraps around from the end of the structure being edited and continues searching from the beginning.

Control-Meta-F Find a specified structure, searching in reverse from the position of the caret.

HELP
Meta-H Show the argument list for the function currently selected, or currently being typed in, in the SEdit prompt window. If the argument list will not fit in the SEdit prompt window, it is displayed in the main Prompt Window.

Meta-I Inspect the current selection.

Meta-J Join any number of sequential Lisp objects of the same type into a single object of that type. Join is supported for atoms, strings, lists, and comments. In addition, SEdit permits joining of a sequence of atoms and strings, since either type can easily be coerced into the other. In this case, the result of the Join will be an atom if the first object in the selection is an atom, otherwise the result will be a string.

Control-L Redisplay the structure being edited.

SKIP-NEXT
Meta-N Select next gap in the structure.

Meta-O Edit the definition of the current selection. If the selected name has more than one type of definition, SEdit asks for the type to edit. If the selection has no definition, a menu pops up. This menu lets you specify the type of definition to create.

Control-Meta-O Perform a fast edit by calling `ED` with the `CURRENT` option.

Meta-P Change the current package for this edit. Prompt the user for a new package name. SEdit will redisplay atoms with respect to that package.

AGAIN
Meta-R Redo the edit change that was just undone. Redo only works directly following an Undo. Any number of Undo commands can be sequentially redone.

SHIFT-FIND

Meta-S Substitute a structure, or sequence of structures within the current selection. SEdit prompts you in the SEdit prompt window for the structures to replace, and the structures to replace with. The selection to substitute within must be a structure selection.

Control-Meta-S Remove all occurences of a structure or sequence of structures within the current selection. SEdit prompts you for the structures to delete.

UNDO
Meta-U Undo the last edit. All changes in the the edit session are remembered, and can be undone sequentially.

Control-W Delete the previous atom or structure. If the caret is in the middle of an atom, deletes backward to the beginning of the atom only.

Control-X Tell SEdit that this session is complete. The SEdit window remains open.

EXPAND
Meta-X Replaces the current selection with its definition. This command can be used to expand macros and translate CLISP.

Control-Meta-X Tell SEdit that this session is complete Close the SEdit window.

Meta-Z Mutate. Prompt for a function and call this function with the current selection as the argument. The result is inserted into SEdit and made the current selection.

For example, you can replace a structure with its value by selecting it and mutating by EVAL.

Meta-; Convert old style comments in the selected structure to new style comments. The converter notices any list that begins with the symbol IL:* as an old style comment. Section 16.1.18, Options, describes the converter options .

Control-Meta-; Put the contents of a structure selection into a comment. This provides an easy way to "comment out" a chunk of code. The Extract command can be used to reverse this process, returning the comment to the structures contained therein.

Meta-/ Extract one level of structure from the current selection. If there is no selection, but there is a structure caret, the list containing the caret is used. This command can be used to strip the parentheses off a list, or to unquote a quoted structure, or to replace a comment with the contained structures.

Meta-'
Meta-`
Meta-,
Meta-.
Meta-@ or **Meta-2**
Meta-# or **Meta-3**
Meta-. Quote the current selection with the specified kind of quote.

Meta-Space
Meta-Return Scroll the current selection to the center of the window. Similarly, the Space or Return key can be used to normalize the caret.

Meta-)
Meta-0 Parenthesize the current selection, position the caret after the new list.

Meta-(
Meta-9 ParenthesizE the current selection, position the caret at the beginning of the new list.

Meta-M Attach a menu of common commands to the top of the SEdit window. Each SEdit window can have its own menu.

SEdit Command Mnemonics

Abort	Meta-A
Change Print Base	Meta-B
Complete	Control-X
Compile & Complete	Control-C
Close, Compile & Complete	Control-Meta-C
Convert Comment	Meta-;
Make Selection Comment	Control-Meta-;
Previous Delete	Control-W
Selection Delete	DELETE
Selection Dot Comma	Meta-.
Selection At Comma	Meta-@
Edit	Meta-O
Fast Edit	Control-Meta-O
Selection Eval	Meta-E
Macro Expand	Meta-X
Forward Find	Meta-F
Reverse Find	Control-Meta-F
Next Gap	Meta-N
Arglist Help	Meta-H
Inspect	Meta-I
Join	Meta-J
Attach Menu	Meta-M
Expression Mutate	Meat-Z
Change Package	Meta-P
Selection Left Parenthesize	Meta-(
Selection Right Parenthesize	Meta-)
Selection Pop	Meta-/
Selection Back Quote	Meta-‘
Selection Hash Quote	Meta-#
Selection Quote	Meta-’
Redisplay	Control-L
Redo	Meta-R
Remove	Control-Meta-S
Substitute	Meta-S
Undo	Meta-U

SEdit Command Menu

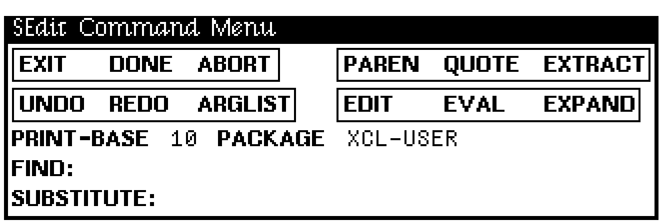
When the mouse cursor is in the SEdit title bar and you press middle mouse button, a Help Menu of commands pops up. The menu looks like this:

Commands	
Abort	M-A
Done	C-X
Done & Compile	C-C
Done & Close	M-C-X
Done, Compile, & Close	M-C-C
Undo	M-U
Redo	M-R
Find	M-F
Reverse Find	M-C-F
Remove	M-C-S
Substitute	M-S
Find Gap	M-N
Arglist	M-H
Convert Comment	M-;
Edit	M-O
Eval	M-E
Expand	M-X
Extract	M-/
Inspect	M-I
Join	M-J
Mutate	M-Z
Parenthesize	M-(
Quote	M-'
Set Print-Base	M-B
Set Package	M-P
Attach Menu	M-M

The Help Menu lists each command and its corresponding Command Key. (C- stands for Control, M- for Meta.) The menu pops up with the mouse cursor next to the last command you used from the menu. This makes it easy to repeat a command.

SEdit Attached Menu

SEdit's Attached Command Menu contains the commonly used commands. Use the Meta-M keyboard command to bring up this menu. The menu can be closed, independently of the SEdit window. The menu looks like:



Menu commands work like the corresponding keyboard commands, except for Find and Substitute.

For Find, SEdit prompts in the menu window, next to the Find button, for the structures to find. Type in the structures then select Find again. The search begins from the caret position in the SEdit window.

Similarly, Substitute prompts next to the Find button for the structures to find, and next to the Substitute button for the structures to replace them with. After both have been typed in, selecting Substitute replaces all occurrences of the Find structures with the Substitute structures, within the current selection.

To selectively substitute, use Find to find the next potential substitution target. If you want to replace it, select Substitute. Otherwise, select Find again to go on.

Selecting either Find or Substitute with the right mouse button erases the old structure to find or substitute from the menu, and prompts for a new one.

SEdit Programmer's Interface

The following sections describe SEdit's programmer's interface. All symbols are external in the package `SEDIT`.

SEdit Window Region Manager

SEdit provides user redefinable functions which control how SEdit chooses the region for a new edit window. In the follwin thext there are a few concepts that you will have to be familiar with. They are:

The region stack. This is a stack of old used regions. The reason to keep these around is that the user probably was comfortable with the old position of the window, so when he starts a new SEdit it is a good bet that he will be happy with the old placement.

SEdit uses the respective value of the symbols `SEDIT::DEFAULT-FONT`, `SEDIT::ITALIC-FONT`, `SEDIT::KEYWORD-FONT`, `SEDIT::COMMENT-FONT`, and `SEDIT::BROKEN-ATOM-FONT` when displaying an expression. The value of these symbols have to be font descriptors.

(**GET-WINDOW-REGION** *context reason name type*) [Function]

This function is called when SEdit wants to know where to place a window it is about to open. This happens whenever the user starts a new SEdit or expands an Sedit icon. The default behavior is to pop a window region off SEdit's stack of regions that have been used in the past. If the stack is empty, SEdit prompts for a new region.

context is the current editor context.

reason is one of `:CREATE` or `:EXPAND` depending on what action prompted the call to `GET-WINDOW-REGION`

name is the name of the structure to be edited.

type is the edit type of the calling context.

(**SAVE-WINDOW-REGION** *context reason name type region*) [Function]

This function is called whenever SEdit is finished with a region and wants to make the region available for other SEdits. This happens whenever an SEdit window is closed or shrunk, or when an SEdit Icon is closed. The default behavior is simply to push the region onto SEdit's stack of regions.

context is the current editor context.

reason is one of :CLOSE, :SHRINK, or :CLOSE-ICON or depending on what action prompted the call to SAVE-WINDOW-REGION

name is the name of the structure to be edited.

type is the edit type of the calling context.

region is the region to be pushed onto the region stack. If region is NIL the old region of the SEdit will be pushed top the region stack.

KEEP-WINDOW-REGION [Variable]

Default T. This flag determines the behavior of the default SEdit region manager, explained above, for shrinking and expanding windows. When set to T, shrinking an SEdit window will not give up that window's region; the icon will always expand back into the same region. When set to NIL, the window's region is made available for other SEdits when the window is shrunk. Then when an SEdit icon is expanded, the window will be reshaped to the next available region.

This variable is only used by the default implementations of the functions get-window-region and save-window-region. If these functions are redefined, this flag is no longer used.

Options

The following parameters can be set as desired.

WRAP-PARENS [Variable]

This SEdit pretty printer flag determines whether or not trailing close parenthesis characters,), are forced to be visible in the window without scrolling. By default it is set to NIL, meaning that close parens are allowed to "fall off" the right edge of the window. If set to T, the pretty printer will start a new line before the structure preceding the close parens, so that all the parens will be visible.

WRAP-SEARCH [Variable]

This flag determines whether or not SEdit find will wrap around to the top of the structure when it reaches the end, or vice versa in the case of reverse find. The default is NIL.

CLEAR-LINEAR-ON-COMPLETION [Variable]

This flag determines whether or not SEdit completely re-pretty prints the structure being edited when you complete the edit. The default value is `NIL`, meaning that SEdit reuses the pretty printing.

IGNORE-CHANGES-ON-COMPLETION [Variable]

Sometimes the structure that you are editing is changed by the system upon completion. Editdates are an example of this behavior. When this flag is `NIL`, the default, SEdit will redisplay the new structure, capturing the changes. When `T`, SEdit will ignore the fact that changes were made by the system and keep the old structure.

CONVERT-UPGRADE [Variable]

Default 100. When using Meta-; to convert old-style single- asterisk comments, if the length of the comment exceeds convert-upgrade characters, the comment is converted into a double semicolon comment. Otherwise, the comment is converted into a single semicolon comment.

Old-style double-asterisk comments are always converted into new-style triple-semicolon comments.

Control Functions

(**RESET**) [Function]

This function recomputes the SEdit edit environment. Any changes made in the font profile, or any changes made to SEdit's commands are captured by resetting. Close all SEdit windows before calling this function.

(**ADD-COMMAND** *key-code form &optional scroll? key-name command-name help-string*) [Function]

This function allows you to write your own SEdit keyboard commands. You can add commands to new keys, or you can redefine keys that SEdit already uses as command keys. If you mistakenly redefine an SEdit command, the funtion Reset-Commands will remove all user-added commands, leaving SEdit with its default set of commands.

key-code can be a character code, or any form acceptible to il:charcode.

form determines the function to be called when the key command is typed. It can be a symbol naming a function, or a list, whose first element is a symbol naming a function and the rest of the elements are extra arguments to the function. When the command is invoked, SEdit will apply the function to the edit context (SEdit's main data structure), the charcode that was typed, and any extra arguments supplied in form. The extra arguments do not get evaluated, but are useful as keywords or flags, depending on how the command was invoked. The command function must return `T` if it handled the command. If the function returns `NIL`, SEdit will ignore the command and insert the character typed.

The first optional argument, *scroll?*, determines whether or not SEdit scrolls the window after running the command. This argument defaults to NIL, meaning don't scroll. If the value of *scroll?* is T, SEdit will scroll the window to ensure that the caret is visible.

The rest of the optional arguments are used to add this command to SEdit's middle button menu. When the item is selected from the menu, the command function will be called as described above, with the charcode argument set to NIL.

key-name is a string to identify the key (combination) to be typed to invoke the command. For example "M-A" to represent the Meta-A key combination, and "C-M-A" for Control-Meta-A.

command-name is a string to identify the command function, and will appear in the menu next to the key-name.

help-string is a string to be printed in the prompt window when a mouse button is held down over the menu item.

After adding all the commands that you want, you must call Reset-Commands to install them.

For example:

```
(ADD-COMMAND "^U" (MY-CHANGE-CASE T))

(ADD-COMMAND "^Y" (MY-CHANGE-CASE NIL))

(ADD-COMMAND "1,R" MY-REMOVE-NIL
  "M-R" "REMOVE NIL"
  "REMOVE NIL FROM THE SELECTED STRUCTURE"))

(RESET-COMMANDS)
```

will add three commands.

Suppose MY-CHANGE-CASE takes the arguments *context, charcode,* and *upper-case?*. *upper-case?* will be set to T when MY-CHANGE-CASE is called from Control-U, and NIL when called from Control-Y. MY-REMOVE-NIL will be called with only *context* and *charcode* arguments when you type Meta-R.

(**RESET-COMMANDS**) [Function]

This function installs all commands added by add-command. SEdits which are open at the time of the reset-commands will not see the new commands; only new SEdits will have the new commands available.

(**DEFAULT-COMMANDS**) [Function]

This function removes all commands added by add-command, leaving SEdit with its default set of commands. As in reset-commands, open SEdits will not be changed; only new SEdits will have the user commands removed.

(**GET-PROMPT-WINDOW** *context*) [Function]

Returns the attached prompt window for a particular SEdit.

(**GET-SELECTION** *context*) [Function]

This function returns two values: the selected structure, and the type of selection, one of NIL, T, or :SUB-LIST. The selection type NIL means there is not a valid selection (in this case the structure is meaningless). T means the selection is one complete structure. :SUB-LIST means a series of elements in a list is selected, in which case the structure returned is a list of the elements selected.

(**REPLACE-SELECTION** *context structure selection-type*) [Function]

This function replaces the current selection with a new structure, or multiple structures, by deleting the selection and then inserting the new structure(s). The selection-type argument must be one of T or :SUB-LIST. If T, the structure is inserted as one complete structure. If :SUB-LIST, the structure is treated as a list of elements, each of which is insertd.

EDIT-FN [Variable]

This function is called with the selected structure and the edit specified as arguments to Sedit options as its arguments from the Edit (M-O) command. It should start the editor as appropriate, or generate an error if the selection is not editable.

COMPILE-FN [Variable]

This function is called with the arguments *name*, *type*, and *body*, from the compile/completion commands. It should compile the definition, *body*, and install the code as appropriate.

(**SEDIT** *structure props options*) [Function]

This function provides a means of starting SEdit directly. *structure* is the structure to be edited.

props is a property list, which may specify the following properties:

:NAME - the name of the object being edited

:TYPE - the file manager type of the object being edited. If NIL, SEdit will not call the file manager when it tries to refetch the definition it is editing. Instead, it will just continue to use the structure that it has.

:COMPLETION-FN - the function to be called when the edit session is completed. This function is called with the *context*, *structure*, and *changed?* arguments. *context* is SEdits main data structure. *structure* is the structure being edited. *changed?* specifies if any changes have been made, and is one of NIL, T, or :ABORT, where :ABORT means the user is aborting the edit and throwing away any changes made. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments for the function.

`:ROOT-CHANGED-FN` - the function to be called when the entire structure being edited is replaced with a new structure. This function is called with the new structure as its argument. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments that the function is applied to following the structure argument.

`options` is one or a list of any number of the followng keywords:

`:CLOSE-ON-COMPLETION` - This option specifies that SEdit cannot remain active for multiple completions. That is, the SEdit window cannot be shrunk, and the completion commands that normally leave the window open will in this case close the window and terminate the edit.

`:COMPILE-ON-COMPLETION` - This option specifies that SEdit should call the `*COMPILE-FN*` to compile the definition being edited upon completion, regardless of the completion command used.

The TTY Editor

This editor the main code editor in pre-window-system versions of Interlisp. For that task, it has been replaced by SEdit.

However, the TTY Editor provides an excellent language for manipulating list structure and making large-scale code changes. For example, several tools for cleaning up code are written using TTY Editor calls to do the actual work.

TTY Editor Local Attention-Changing Commands

This section describes commands that change the current expression (i.e., change the edit chain) thereby "shifting the editor's attention." These commands depend only on the *structure* of the edit chain, as compared to the search commands (presented later), which search the contents of the structure.

`UP` [Editor Command]

`UP` modifies the edit chain so that the old current expression (i.e., the one at the time `UP` was called) is the first element in the new current expression. If the current expression is the first element in the next higher expression `UP` simply does a `0`. Otherwise `UP` adds the corresponding tail to the edit chain.

If a `P` command would cause the editor to type `...` before typing the current expression, ie., the current expression is a tail of the next higher expression, `UP` has no effect.

For example:

```
*PP
(COND ((NULL X) (RETURN Y)))
*1 P
COND
*UP P
(COND (& &))
*-1 P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))
*F NULL P
(NULL X)
*UP P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))
```

The execution of UP is straightforward, except in those cases where the current expression appears more than once in the next higher expression. For example, if the current expression is (A NIL B NIL C NIL) and you perform 4 followed by UP, the current expression should then be ... NIL C NIL). UP can determine which tail is the correct one because the commands that descend save the last tail on an internal editor variable, LASTAIL. Thus after the 4 command is executed, LASTAIL is (NIL C NIL). When UP is called, it first determines if the current expression is a tail of the next higher expression. If it is, UP is finished. Otherwise, UP computes (MEMB *CURRENT-EXPRESSION NEXT-HIGHER-EXPRESSION*) to obtain a tail beginning with the current expression. The current expression should *always* be either a tail or an element of the next higher expression. If it is neither, for example yopu have directly (and incorrectly) manipulated the edit chain, UP generates an error. If there are no other instances of the current expression in the next higher expression, this tail is the correct one. Otherwise UP uses LASTAIL to select the correct tail.

Occasionally you can get the edit chain into a state where LASTAIL cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were EQ, and you descended more than one level into one of them and then tried to come back out using UP. In this case, UP prints LOCATION UNCERTAIN and generates an error. Of course, we could have solved this problem completely in our implementation by saving at each descent *both* elements and tails. However, this would be a costly solution to a situation that arises infrequently, and when it does, has no detrimental effects. The LASTAIL solution is cheap and resolves almost all of the ambiguities.

N (*N*> = *1*) [Editor Command]

Adds the *N*th element of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets LASTAIL for use by UP. Generates an error if the current expression is not a list that contains at least *N* elements.

-*N* (*N*> = 1) [Editor Command]

Adds the *N*th element from the end of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets LASTAIL for use by UP. Generates an error if the current expression is not a list that contains at least *N* elements.

0 [Editor Command]

Sets the edit chain to CDR of the edit chain, thereby making the next higher expression be the new current expression. Generates an error if there is no higher expression, i.e., CDR of edit chain is NIL.

Note that 0 usually corresponds to going back to the next higher left parenthesis, but not always. For example:

```
*P
(A B C D E F B)
*3 UP P
... C D E F G)
*3 UP P
... E F G)
*0 P
... C D E F G)
```

If the intention is to go back to the next higher left parenthesis, regardless of any intervening tails, the command !0 can be used.

!0 [Editor Command]

Does repeated 0's until it reaches a point where the current expression is *not* a tail of the next higher expression, i.e., always goes back to the next higher left parenthesis.

↑ [Editor Command]

Sets the edit chain to LAST of edit chain, thereby making the top level expression be the current expression. Never generates an error.

NX [Editor Command]

Effectively does an UP followed by a 2, thereby making the current expression be the next expression. Generates an error if the current expression is the last one in a list. (However, !NX described below will handle this case.)

BK [Editor Command]

Makes the current expression be the previous expression in the next higher expression. Generates an error if the current expression is the first expression in a list.

For example:

```
*PP
(COND ((NULL X) (RETURN Y)))
*F RETURN P
(RETURN Y)
*BK P
(NULL X)
```

Both NX and BK operate by performing a !0 followed by an appropriate number, i.e., there won't be an extra tail above the new current expression, as there would be if NX operated by performing an UP followed by a 2.

(NX *N*) [Editor Command]

(*N* >= 1) Equivalent to *N* NX commands, except if an error occurs, the edit chain is not changed.

(BK *N*) [Editor Command]

(*N* >= 1) Equivalent to *N* BK commands, except if an error occurs, the edit chain is not changed.

Note: (NX -*N*) is equivalent to (BK *N*), and vice versa.

!NX [Editor Command]

Makes the current expression be the next expression at a higher level, i.e., goes through any number of right parentheses to get to the next expression. For example:

```
*PP
(PROG ((L L)
       (UF L))
  LP  (COND
         ((NULL (SETQ L (CDR L)))
            (ERROR!))
         ([NULL (CDR (FMEMB (CAR L) (CADR L]
            (GO LP)))
      (EDITCOM (QUOTE NX))
      (SETQ UNFIND UF)
      (RETURN L))
*F CDR P
(CDR L)
*NX

NX   ?
*!NX P
(ERROR!)
*!NX P
((NULL &) (GO LP))
*!NX P
(EDITCOM (QUOTE NX))
*
```

`!NX` operates by doing `0`'s until it reaches a stage where the current expression is *not* the last expression in the next higher expression, and then does a `NX`. Thus `!NX` always goes through at least one unmatched right parenthesis, and the new current expression is always on a different level, i.e., `!NX` and `NX` always produce different results. For example using the previous current expression:

```
*F CAR P
(CAR L)
*!NX P
(GO LP)
*\P P
(CAR L)
*NX P
(CADR L)
*
```

`(NTH` *`N`*`)` [Editor Command]

(*N* `~= 0`) Equivalent to *N* followed by `UP`, i.e., causes the list starting with the *N*th element of the current expression (or *N*th from the end if *N* < 0) to become the current expression. Causes an error if current expression does not have at least *N* elements.

`(NTH 1)` is a no-op, as is `(NTH -`*`L`*`)` where *L* is the length of the current expression.

`line-feed` [Editor Command]

Moves to the "next" expression and prints it, i.e. performs a `NX` if possible, otherwise performs a `!NX`. (The latter case is indcated by first printing ">".)

`Control-X` [Editor Command]

Control-X moves to the "previous" thing and then prints it, i.e. performs a `BK` if possible, otherwise a `!0` followed by a `BK`.

`Control-Z` [Editor Command]

Control-Z moves to the last expression and prints it, i.e. does `-1` followed by `P`.

Line-feed, Control-X, and Control-Z are implemented as *immediate* read macros; as soon as they are read, they abort the current printout. They thus provide a convenient way of moving around in the editor. To facilitate using different control characters for those macros, the function `SETTERMCHARS` is provided (see below).

Commands That Search

All of the editor commands that search use the same pattern matching routine (the function EDIT4E, below). We will therefore begin our discussion of searching by describing the pattern match mechanism. A pattern *PAT* matches with *X* if any of the following conditions are true:

1. If *PAT* is EQ to *X*
2. If *PAT* is &
3. If *PAT* is a number and EQP to *X*
4. If *PAT* is a string and (STREQUAL *PAT X*) is true
5. If (CAR *PAT*) is the atom *ANY*, (CDR *PAT*) is a list of patterns, and one of the patterns on (CDR *PAT*) matches *X*.
6. If *PAT* is a literal atom or string containing one or more $s (escapes), each $ can match an indefinite number (including 0) of contiguous characters in the atom or string *X*, e.g., VER$ matches both VERYLONGATOM and "VERYLONGSTRING" as do $LONG$ (but not $LONG), and VLT. Note: the litatom $ (escape) matches only with itself.
7. If *PAT* is a literal atom or string ending in $$ (escape, escape), *PAT* matches with the atom or string *X* if it is "close" to *PAT*, in the sense used by the spelling corrector (see Chapter 20). For example, CONSS$$ matches with CONS, CNONC$$ with NCONC or NCONC1.

The pattern matching routine always types a message of the form =*MATCHING-ITEM* to inform you of the object matched by a pattern of the above two types, unless EDITQUIETFLG = T. For example, if VER$ matches VERYLONGATOM, the editor would print =VERYLONGATOM.

8. If (CAR *PAT*) is the atom **--**, *PAT* matches *X* if (CDR *PAT*) matches with some tail of *X*. For example, (A -- (&)) will match with (A B C (D)), but not (A B C D), or (A B C (D) E). However, note that (A -- (&) --) will match with (A B C (D) E). In other words, -- can match any interior segment of a list.

   If (CDR *PAT*) = NIL, i.e., *PAT* = (--), then it matches any tail of a list. Therefore, (A --) matches (A), (A B C) and (A . B).

9. If (CAR *PAT*) is the atom ==, *PAT* matches *X* if and only if (CDR *PAT*) is EQ to *X*.

   This pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command *typed* in by you obviously cannot be EQ to already existing structure.

10. If (CADR *PAT*) is the atom .. (two periods), *PAT* matches *X* if (CAR *PAT*) matches (CAR *X*) and (CDDR *PAT*) is contained in *X*, as described below.
11. Otherwise if *X* is a list, *PAT* matches *X* if (CAR *PAT*) matches (CAR *X*), and (CDR *PAT*) matches (CDR *X*).

When the editor is searching, the pattern matching routine is called to match with *elements* in the structure, unless the pattern begins with ... (three periods), in which case CDR of the pattern is matched against proper tails in the structure. Thus,

```
*P
         (A B C (B C))
         *F (B --)
         *P
         (B C)
         *0 F (... B --)
         *P
         ... B C (B C))
```

Matching is also attempted with atomic tails (except for NIL). Thus,

```
*P
         (A (B . C))
         *F C
         *P
         ... . C)
```

Although the current expression is the atom C after the final command, it is printed as ... . C) to alert you to the fact that C is a *tail*, not an element. Note that the pattern C will match with either instance of C in (A C (B . C)), whereas (... . C) will match only the second C. The pattern NIL will only match with NIL as an element, i.e., it will not match in (A B), even though CDDR of (A B) is NIL. However, (... . NIL) (or equivalently (...)) may be used to specify a NIL *tail*, e.g., (... . NIL) will match with CDR of the third subexpression of ((A . B) (C . D) (E)).

Search Algorithm

Searching begins with the current expression and proceeds in print order. Searching usually means find the next instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged. At each step, the pattern is matched against the next element in the expression currently being searched, unless the pattern begins with ... (three periods) in which case it is matched against the next tail of the expression.

If the match is not successful, the search operation is recursive first in the CAR direction, and then in the CDR direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level. Note: A find command of the form (F *PATTERN* NIL) will only attempts matches at the top level of the current expression, i.e., it does not descend into elements, or ascend to higher expressions.

However, at no point is the total recursive depth of the search (sum of number of CARs and CDRs descended into) allowed to exceed the value of the variable MAXLEVEL. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the element or tail for which the recursive depth is below MAXLEVEL. This feature is designed to enable you to search circular list structures (by setting MAXLEVEL small), as well as protecting him from accidentally encountering a circular list structure in the course of normal editing. MAXLEVEL can also be set to NIL, which is equivalent to infinity. MAXLEVEL is initially set to 300.

If a successful match is not found in the current expression, the search automatically ascends to the next higher expression, and continues searching there on the next expression after the expression it just finished searching. If there is none, it ascends again, etc. This process continues until the entire edit chain has been searched, at which point the search fails, and an error is generated. If the search fails (or is aborted by Control-E), the edit chain is not changed (nor are any CONSes performed).

If the search is successful, i.e., an expression is found that the pattern matches, the edit chain is set to the value it would have had had you reached that expression via a sequence of integer commands.

If the expression that matched was a list, it will be the final link in the edit chain, i.e., the new current expression. If the expression that matched is not a list, e.g., is an atom, the current expression will be the tail beginning with that atom, unless the atom is a tail, e.g., B in (A . B). In this case, the current expression will be B, but will print as ... . B). In other words, the search effectively does an UP (unless UPFINDFLG = NIL (initially T). See "Form Oriented Editing" in this chapter).

Search Commands

All of the commands below set LASTAIL for use by UP, set UNFIND for use by \ (below), and do not change the edit chain or perform any CONSes if they are unsuccessful or aborted.

F *PATTERN* [Editor Command]

Actually two commands: the F informs the editor that the *next* command is to be interpreted as a pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., F *PATTERN* means find the next instance of *PATTERN*.

If (MEMB *PATTERN* *CURRENT-EXPRESSION*) is true, F does not proceed with a full recursive search. If the value of the MEMB is NIL, F invokes the search algorithm described above.

If the current expression is (PROG NIL LP (COND (-- (GO LP1))) ... LP1 ...), then F LP1 will find the PROG label, not the LP1 inside of the GO expression, even though the latter appears first (in print order) in the current expression. Typing 1 (making the atom PROG be the current expression) followed by F LP1 *would* find the first LP1.

F *PATTERN* N [Editor Command]

Same as F *PATTERN*, i.e., Finds the Next instance of *PATTERN*, except that the MEMB check of F *PATTERN* is not performed.

F *PATTERN* T [Editor Command]

Similar to F *PATTERN*, except that it may succeed without changing the edit chain, and it does not perform the MEMB check. For example, if the current expression is (COND ...), F COND will look for the next COND, but (F COND T) will "stay here".

(F *PATTERN* N) [Editor Command]

(*N* >= 1) Finds the *N*th place that *PATTERN* matches. Equivalent to (F *PATTERN* T) followed by (F *PATTERN* N) repeated *N*-1 times. Each time *PATTERN* successfully matches, *N* is decremented by 1, and the search continues, until *N* reaches 0. Note that *PATTERN* does not have to match with *N* identical expressions; it just has to match *N* times. Thus if the current expression is (FOO1 FOO2 FOO3), (F FOO$ 3) will find FOO3.

If *PATTERN* does not match successfully *N* times, an error is generated and the edit chain is unchanged (even if *PATTERN* matched *N*-1 times).

(F *PATTERN*) [Editor Command]
F *PATTERN* NIL [Editor Command]

Similar to F *PATTERN*, except that it only matches with elements at the top level of the current expression, i.e., the search will not descend into the current expression, nor will it go outside of the current expression. May succeed without changing the edit chain.

For example, if the current expression is (PROG NIL (SETQ X (COND & &)) (COND &) ...), the command F COND will find the COND inside the SETQ, whereas (F (COND --)) will find the top level COND, i.e., the second one.

(FS $PATTERN_1$ ... $PATTERN_N$) [Editor Command]

Equivalent to F $PATTERN_1$ followed by F $PATTERN_2$ ... followed by F $PATTERN_N$, so that if F $PATTERN_M$ fails, the edit chain is left at the place $PATTERN_{M-1}$ matched.

(F= *EXPRESSION* *X*) [Editor Command]

Equivalent to (F (== . *EXPRESSION*) *X*), i.e., searches for a structure EQ to *EXPRESSION* (see above).

(ORF $PATTERN_1$ ... $PATTERN_N$) [Editor Command]

Equivalent to (F (*ANY* $PATTERN_1$ ... $PATTERN_N$) N), i.e., searches for an expression that is matched by either $PATTERN_1$, $PATTERN_2$, ... or $PATTERN_N$ (see above).

BF *PATTERN* [Editor Command]

"Backwards Find". Searches in reverse print order, beginning with the expression immediately before the current expression (unless the current expression is the top level expression, in which case BF searches the entire expression, in reverse order).

BF uses the same pattern match routine as F, and MAXLEVEL and UPFINDFLG have the same effect, but the searching begins at the *end* of each list, and descends into each element before attempting to match that element. If unsuccessful, the search continues with the next previous element, etc., until the front of the list is reached, at which point BF ascends and backs up, etc.

For example, if the current expression is

```
(PROG NIL (SETQ X (SETQ Y (LIST Z))) (COND ((SETQ W --) --)) --),
```

the command F LIST followed by BF SETQ will leave the current expression as (SETQ Y (LIST Z)), as will F COND followed by BF SETQ.

BF *PATTERN* T [Editor Command]

Similar to BF *PATTERN*, except that the search always includes the current expression, i.e., starts at the end of current expression and works backward, then ascends and backs up, etc.

Thus in the previous example, where F COND followed by BF SETQ found (SETQ Y (LIST Z)), F COND followed by (BF SETQ T) would find the (SETQ W --) expression.

(BF *PATTERN*) [Editor Command]
BF *PATTERN* NIL [Editor Command]

Same as BF *PATTERN*.

(GO *LABEL*) [Editor Command]

Makes the current expression be the first thing after the PROG label *LABEL*, i.e. goes where an executed GO would go.

Location Specification

Many of the more sophisticated commands described later in this chapter use a more general method of specifying position called a "location specification." A location specification is a list of edit commands that are executed in the normal fashion with two exceptions. First, all commands not recognized by the editor are interpreted as though they had been preceded by F; normally such commands would cause errors. For example, the location specification (COND 2 3) specifies the 3rd element in the first clause of the next COND. Note that you could always write F COND followed by 2 and 3 for (COND 2 3) if you were not sure whether or not COND was the name of an atomic command.

Secondly, if an error occurs while evaluating one of the commands in the location specification, and the edit chain had been changed, i.e., was not the same as it was at the beginning of that execution of the location specification, the location operation will continue. In other words, the location operation keeps going unless it reaches a state where it detects that it is "looping", at which point it gives up. Thus, if (COND 2 3) is being located, and the first clause of the next COND contained only two elements, the execution of the command 3 would cause an error. The search would then continue by looking for the next COND. However, if a point were reached where there were no further CONDs, then the first command, COND, would cause the error; the edit chain would not have been changed, and so the entire location operation would fail, and cause an error.

The IF command (see above) in conjunction with the ## function (see below) provide a way of using arbitrary predicates applied to elements in the current expression. IF and ## will be described in detail later in the chapter, along with examples illustrating their use in location specifications.

Throughout this chapter, the meta-symbol @ is used to denote a location specification. Thus @ is a list of commands interpreted as described above. @ can also be atomic, in which case it is interpreted as (LIST @).

(LC . @) [Editor Command]

Provides a way of explicitly invoking the location operation, e.g., (LC COND 2 3) will perform the the search described above.

(LCL . @) [Editor Command]

Same as LC except the search is confined to the current expression, i.e., the edit chain is rebound during the search so that it looks as though the editor were called on just the current expression. For example, to find a COND containing a RETURN, one might use the location specification (COND (LCL RETURN) \) where the \ would reverse the effects of the LCL command, and make the final current expression be the COND.

(2ND . @) [Editor Command]

Same as (LC . @) followed by another (LC . @) except that if the first succeeds and second fails, no change is made to the edit chain.

(3ND . @) [Editor Command]

Similar to 2ND.

(← *PATTERN*) [Editor Command]

Ascends the edit chain looking for a link which matches *PATTERN*. In other words, it keeps doing 0's until it gets to a specified point. If *PATTERN* is atomic, it is matched with the first element of each link, otherwise with the entire link. If no match is found, an error is generated, and the edit chain is unchanged.

If *PATTERN* is of the form (IF *EXPRESSION*), *EXPRESSION* is evaluated at each link, and if its value is NIL, or the evaluation causes an error, the ascent continues.

For example:

```
*PP
[PROG NIL
   (COND
      [(NULL (SETQ L (CDR L)))
        (COND
           (FLG (RETURN L]
      ([NULL (CDR (FMEMB (CAR L)
        (CADR L]]
*F CADR
*(← COND)
*P
(COND (& &) (& &))
*
```

Note that this command differs from BF in that it does not search *inside* of each link, it simply ascends. Thus in the above example, F CADR followed by BF COND would find (COND (FLG (RETURN L))), not the higher COND.

(BELOW *COM X*) [Editor Command]

Ascends the edit chain looking for a link specified by *COM*, and stops *X* links below that (only links that are elements are counted, not tails). In other words BELOW keeps doing 0's until it gets to a specified point, and then backs off *X* 0's.

Note that *X* is evaluated, so one can type (BELOW *COM* (IPLUS X Y)).

(BELOW *COM*) [Editor Command]

Same as (BELOW *COM* 1).

For example, (BELOW COND) will cause the COND *clause* containing the current expression to become the new current expression. Thus if the current expression is as shown above, F CADR followed by (BELOW COND) will make the new expression be ([NULL (CDR (FMEMB (CAR L) (CADR L] (GO LP)), and is therefore equivalent to 0 0 0 0.

The BELOW command is useful for locating a substructure by specifying something it contains. For example, suppose you are editing a list of lists, and want to find a sublist that contains a FOO (at any depth). He simply executes F FOO (BELOW \).

(NEX *COM*) [Editor Command]

Same as (BELOW *COM*) followed by NX.

For example, if you are deep inside of a SELECTQ clause, you can advance to the next clause with (NEX SELECTQ).

NEX [Editor Command]

Same as (NEX ←).

The atomic form of NEX is useful if you will be performing repeated executions of (NEX *COM*). By simply MARKing (see the next section) the chain corresponding to *COM*, you can use NEX to step through the sublists.

(NTH *COM*) [Editor Command]

Generalized NTH command. Effectively performs (LCL . *COM*), followed by (BELOW \), followed by UP.

If the search is unsuccessful, NTH generates an error and the edit chain is not changed.

Note that (NTH *NUMBER*) is just a special case of (NTH *COM*), and in fact, no special check is made for *COM* a number; both commands are executed identically.

In other words, NTH locates *COM*, using a search restricted to the current expression, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation. For example:

```
*P
(PROG (& &) LP (COND & &) (EDITCOM &) (SETQ UNFIND
UF) (RETURN L))
*(NTH UF)
*P
... (SETQ UNFIND UF) (RETURN L))
*
```

PATTERN .. @ [Editor Command]

For example, (COND .. RETURN). Finds a COND that contains a RETURN, at any depth. Equivalent to (but more efficient than) (F *PATTERN* N), (LCL . @) followed by (← *PATTERN*).

An infix command, "`..`" is not a meta-symbol, it *is* the name of the command. `@` is `CDDR` of the command. Note that `(`*`PATTERN`*` .. @)` can also be used directly as an edit pattern as described above, e.g. `F (`*`PATTERN`*` .. @)`.

For example, if the current expression is

```
(PROG NIL [COND ((NULL L) (COND (FLG (RETURN L] --),
```

then `(COND .. RETURN)` will make `(COND (FLG (RETURN L)))` be the current expression. Note that it is the innermost `COND` that is found, because this is the first `COND` encountered when ascending from the `RETURN`. In other words, `(`*`PATTERN`*` .. @)` is not *always* equivalent to `(F `*`PATTERN`*` N)`, followed by `(LCL . @)` followed by `\`.

Note that `@` is a location specification, not just a pattern. Thus `(RETURN .. COND 2 3)` can be used to find the `RETURN` which contains a `COND` whose first clause contains (at least) three elements. Note also that since **@** permits any edit command, you can write commands of the form `(COND .. (RETURN .. COND))`, which will locate the first `COND` that contains a `RETURN` that contains a `COND`.

Commands That Save and Restore the Edit Chain

Several facilities are available for saving the current edit chain and later retrieving it: `MARK`, which marks the current chain for future reference, ←, which returns to the last mark without destroying it, and ←←, which returns to the last mark and also erases it.

`MARK` [Editor Command]

Adds the current edit chain to the front of the list `MARKLST`.

← [Editor Command]

Makes the new edit chain be `(CAR MARKLST)`. Generates an error if `MARKLST` is `NIL`, i.e., no `MARK`s have been performed, or all have been erased.

This is an atomic command; do not confuse it with the list command `(← `*`PATTERN`*`)`.

←← [Editor Command]

Similar to ← but also erases the last `MARK`, i.e., performs `(SETQ MARKLST (CDR MARKLST))`.

If you have two chains marked, and wish to return to the first chain, you must perform ←←, which removes the second mark, and then ←. However, the second mark is then no longer accessible. If you want to be able to return to either of two (or more) chains, you can use the following generalized MARK:

(MARK *SYMBOL*) [Editor Command]

Sets *SYMBOL* to the current edit chain,

(\ *SYMBOL*) [Editor Command]

Makes the current edit chain become the value of *SYMBOL*.

If you did not prepare in advance for returning to a particular edit chain, you may still be able to return to that chain with a single command by using \ or \P.

\ [Editor Command]

Makes the edit chain be the value of UNFIND. Generates an error if UNFIND = NIL.

UNFIND is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ↑, ←, ←←, !NX, all commands that involve a search, e.g., F, LC, .., BELOW, et al and \ and \P themselves. One exception is that UNFIND is not reset when the current edit chain is the top level expression, since this could always be returned to via the ↑ command.

For example, if you type F COND, and then F CAR, \ would take you back to the COND. Another \ would take you back to the CAR, etc.

\P [Editor Command]

Restores the edit chain to its state as of the last print operation, i.e., P, ?, or PP. If the edit chain has not changed since the last printing, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

For example, if you type P followed by 3 2 1 P, \P returns to the first P, i.e., would be equivalent to 0 0 0. Another \P would then take you back to the second P, i.e., you could use \P to flip back and forth between the two edit chains.

If you had typed P followed by F COND, you could use *either* \ or \P to return to the P, i.e., the action of \ and \P are independent.

S *SYMBOL* @ [Editor Command]

Sets *SYMBOL* (using SETQ) to the current expression after performing (LC . @). The edit chain is not changed.

Thus `(S FOO)` will set `FOO` to the current expression, and `(S FOO -1 1)` will set `FOO` to the first element in the last element of the current expression.

Commands That Modify Structure

The basic structure modification commands in the editor are:

`(`*N*`)` `(`*N* `>= 1)` [Editor Command]

Deletes the corresponding element from the current expression.

`(`*N* E_1 `...` E_M`)` `(`*N* `>= 1)` [Editor Command]

Replaces the *N*th element in the current expression with E_1 `...` E_M.

`(-`*N* E_1 `...` E_M`)` `(`*N* `>= 1)` [Editor Command]

Inserts E_1 `...` E_M before the *N*th element in the current expression.

`(N` E_1 `...` E_M`)` [Editor Command]

Attaches E_1 `...` E_M at the end of the current expression.

As mentioned earlier: *all structure modification done by the editor is destructive, i.e., the editor uses* `RPLACA` *and* `RPLACD` *to physically change the structure it was given.* However, all structure modification is undoable, see `UNDO` .

All of the above commands generate errors if the current expression is not a list, or in the case of the first three commands, if the list contains fewer than *N* elements. In addition, the command `(1)`, i.e., delete the first element, will cause an error if there is only one element, since deleting the first element must be done by replacing it with the second element, and then deleting the second element. Or, to look at it another way, deleting the first element when there is only one element would require changing a list to an atom (i.e., to `NIL`) which cannot be done. However, the command `DELETE` will work even if there is only one element in the current expression, since it will ascend to a point where it *can* do the deletion.

If the value of `CHANGESARRAY` is a hash array, the editor will mark all structures that are changed by doing `(PUTHASH` *STRUCTURE FN* `CHANGESARRAY)`, where *FN* is the name of the function. The algorithm used for marking is as follows:

1. If the expression is inside of another expression already marked as being changed, do nothing.

2. If the change is an insertion of or replacement with a list, mark the list as changed.

3. If the change is an insertion of or replacement with an atom, or a deletion, mark the parent as changed.

CHANGESARRAY is primarily for use by PRETTYPRINT (Chapter 26). When the value of CHANGECHAR is non-NIL, PRETTYPRINT, when printing to a file or display terminal, prints CHANGECHAR in the right margin while printing an expression marked as having been changed. CHANGECHAR is initially |.

Implementation

> *Note: Since all commands that insert, replace, delete or attach structure use the same low level editor functions, the remarks made here are valid for all structure changing commands.*

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor, *copies* of the corresponding structure are used, because of the possibility that the exact same command, (i.e., same list structure) might be used again. Thus if a program constructs the command (1 (A B C)) e.g., via (LIST 1 FOO), and gives this command to the editor, the (A B C) used for the replacement will *not* be EQ to FOO. You can circumvent this by using the I command, which computes the structure to be used. In the above example, the form of the command would be (I 1 FOO), which would replace the first element with the value of FOO itself.

> Note: Some editor commands take as arguments a list of edit commands, e.g., (LP F FOO (1 (CAR FOO))). In this case, the command (1 (CAR FOO)) is not considered to have been "typed in" even though the LP command itself may have been typed in. Similarly, commands originating from macros, or commands given to the editor as arguments to EDITF, EDITV, et al, e.g., EDITF(FOO F COND (N --)) are not considered typed in.

The rest of this section is included for applications wherein the editor is used to modify a data structure, and pointers into that data structure are stored elsewhere. In these cases, the actual mechanics of structure modification must be known in order to predict the effect that various commands may have on these outside pointers. For example, if the value of FOO is CDR of the current expression, what will the commands (2), (3), (2 X Y Z), (-2 X Y Z), etc. do to FOO?

Deletion of the first element in the current expression is performed by replacing it with the second element and deleting the second element by patching around it. Deletion of any other element is done by patching around it, i.e., the previous tail is altered. Thus if FOO is EQ to the current expression which is (A B C D), and FIE is CDR of FOO, after executing the command (1), FOO will be (B C D) (which is EQUAL but not EQ to FIE). However, under the same initial conditions, after executing (2) FIE will be unchanged, i.e., FIE will still be (B C D) even though the current expression and FOO are now (A C D).

A general solution of the problem isn't possible, as it would require being able to make two lists EQ to each other that were originally different. Thus if FIE is CDR of the current expression, and FUM is CDDR of the current expression, performing (2) would have to make FIE be EQ to FUM if all subsequent operations were to update both FIE and FUM correctly.

Both replacement and insertion are accomplished by smashing both CAR and CDR of the corresponding tail. Thus, if FOO were EQ to the current expression, (A B C D), after (1 X Y Z), FOO would be (X Y Z B C D). Similarly, if FOO were EQ to the current expression, (A B C D), then after (-1 X Y Z), FOO would be (X Y Z A B C D).

The N command is accomplished by smashing the last CDR of the current expression a la NCONC. Thus if FOO were EQ to any tail of the current expression, after executing an N command, the corresponding expressions would also appear at the end of FOO.

In summary, the only situation in which an edit operation will *not* change an external pointer occurs when the external pointer is to a *proper tail* of the data structure, i.e., to CDR of some node in the structure, and the operation is deletion. If all external pointers are to *elements* of the structure, i.e., to CAR of some node, or if only insertions, replacements, or attachments are performed, the edit operation will *always* have the same effect on an external pointer as it does on the current expression.

The A, B, and : Commands

In the (*N*), (*N* E_1 ... E_M), and (-*N* E_1 ... E_M) commands, the sign of the integer is used to indicate the operation. As a result, there is no direct way to express insertion after a particular element (hence the necessity for a separate N command). Similarly, you cannot specify deletion or replacement of the *N*th element from the end of a list without first converting *N* to the corresponding positive integer. Accordingly, we have:

(B E_1 ... E_M) [Editor Command]

Inserts E_1 ... E_M before the current expression. Equivalent to UP followed by (-1 E_1 ... E_M).

For example, to insert FOO before the last element in the current expression, perform -1 and then (B FOO).

(A E_1 ... E_M) [Editor Command]

Inserts E_1 ... E_M after the current expression. Equivalent to UP followed by (-2 E_1 ... E_M) or (N E_1 ... E_M), whichever is appropriate.

(: E_1 ... E_M) [Editor Command]

Replaces the current expression by E_1 ... E_M. Equivalent to UP followed by (1 E_1 ... E_M).

DELETE [Editor Command]

(:) [Editor Command]

Deletes the current expression.

DELETE first tries to delete the current expression by performing an UP and then a (1). This works in most cases. However, if after performing UP, the new current expression contains only one element, the command (1) will not work. Therefore, DELETE starts over and performs a BK, followed by UP, followed by (2). For example, if the current expression is (COND ((MEMB X Y)) (T Y)), and you perform -1, and then DELETE, the BK-UP-(2) method is used, and the new current expression will be ... ((MEMB X Y))).

However, if the next higher expression contains only one element, BK will not work. So in this case, DELETE performs UP, followed by (: NIL), i.e., it *replaces* the higher expression by NIL. For example, if the current expression is (COND ((MEMB X Y)) (T Y)) and you perform F MEMB and then DELETE, the new current expression will be ... NIL (T Y)) and the original expression would now be (COND NIL (T Y)). The rationale behind this is that deleting (MEMB X Y) from ((MEMB X Y)) changes a list of one element to a list of no elements, i.e., () or NIL.

If the current expression is a tail, then B, A, :, and DELETE all work exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (B (PRINT X)) would insert (PRINT X) before (PRINT Y), leaving the current expression ... (PRINT X) (PRINT Y) (PRINT Z)).

The following forms of the A, B, and : commands incorporate a location specification:

(INSERT E_1 ... E_M BEFORE . @) [Editor Command]

(@ is (CDR (MEMBER 'BEFORE *COMMAND*))) Similar to (LC .@) followed by (B E_1 ... E_M).

Warning: If @ causes an error, the location process does *not* continue as described above. For example, if @ = (COND 3) and the next COND does not have a thirdelement, the search stops and the INSERT fails. You can always write (LC COND 3) if you intend the search to continue.

```
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR &
&) (PRIN1 & T)
(PRIN1 & T) (SETQ X &

*(INSERT LABEL BEFORE PRIN1)
*P
```

```
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR &
&) LABEL
(PRIN1 & T) (         user typed Control-E

*
```

Current edit chain is not changed, but UNFIND is set to the edit chain after the B was performed, i.e., \ will make the edit chain be that chain where the insertion was performed.

(INSERT E_1 ... E_M AFTER . @) [Editor Command]

Similar to INSERT BEFORE except uses A instead of B.

(INSERT E_1 ... E_M FOR . @) [Editor Command]

Similar to INSERT BEFORE except uses : for B.

(REPLACE @ BY E_1 ... E_M) [Editor Command]

(REPLACE @ WITH E_1 ... E_M) [Editor Command]

Here @ is the *segment* of the command between REPLACE and WITH. Same as (INSERT E_1 ... E_M FOR . @).

Example: (REPLACE COND -1 WITH (T (RETURN L)))

(CHANGE @ TO E_1 ... E_M) [Editor Command]

Same as REPLACE WITH.

(DELETE . @) [Editor Command]

Does a (LC . @) followed by DELETE (see warning about INSERT above). The current edit chain is not changed, but UNFIND is set to the edit chain after the DELETE was performed.

> Note: The edit chain will be changed if the current expression is no longer a part of the expression being edited, e.g., if the current expression is ... C) and you perform (DELETE 1), the tail, (C), will have been cut off. Similarly, if the current expression is (CDR Y) and you perform (REPLACE WITH (CAR X)).

Example: (DELETE -1), (DELETE COND 3)

> *Note: If @ is NIL (i.e., empty), the corresponding operation is performed on the current edit chain.*

For example, `(REPLACE WITH (CAR X))` is equivalent to `(: (CAR X))`. For added readability, `HERE` is also permitted, e.g., `(INSERT (PRINT X) BEFORE HERE)` will insert `(PRINT X)` before the current expression (but not change the edit chain).

Note: *`@` does not have to specify a location within the current expression, i.e., it is perfectly legal to ascend to `INSERT`, `REPLACE`, or `DELETE`*

For example, `(INSERT (RETURN) AFTER ^ PROG -1)` will go to the top, find the first `PROG`, and insert a `(RETURN)` at its end, and not change the current edit chain.

The `A`, `B`, and `:` commands, commands, (and consequently `INSERT`, `REPLACE`, and `CHANGE`), all make special checks in E_1 thru E_M for expressions of the form `(## . COMS)`. In this case, the expression used for inserting or replacing is a *copy* of the current expression after executing *COMS*, a list of edit commands (the execution of *COMS* does not change the current edit chain). For example, `(INSERT (## F COND -1 -1) AFTER 3)` will make a copy of the last form in the last clause of the next `COND`, and insert it after the third element of the current expression. Note that this is not the same as `(INSERT F COND -1 (## -1) AFTER 3)`, which inserts four elements after the third element, namely `F`, `COND`, `-1`, and a copy of the last element in the current expression.

Form Oriented Editing and the Role of UP

The `UP` that is performed before `A`, `B`, and `:` commands (and therefore in `INSERT`, `CHANGE`, `REPLACE`, and `DELETE` commands after the location portion of the operation has been performed) makes these operations form-oriented. For example, if you type `F SETQ`, and then `DELETE`, or simply `(DELETE SETQ)`, you will delete the entire `SETQ` expression, whereas `(DELETE X)` if `X` is a variable, deletes just the variable `X`. In both cases, the operation is performed on the corresponding *form*, and in both cases is probably what you intended. Similarly, if you type `(INSERT (RETURN Y) BEFORE SETQ)`, you mean before the `SETQ` expression, not before the atom `SETQ`. A consequent of this procedure is that a pattern of the form `(SETQ Y --)` can be viewed as simply an elaboration and further refinement of the pattern `SETQ`. Thus `(INSERT (RETURN Y) BEFORE SETQ)` and `(INSERT (RETURN Y) BEFORE (SETQ Y --))` perform the same operation (assuming the next `SETQ` is of the form `(SETQ Y --)`) and, in fact, this is one of the motivations behind making the current expression after `F SETQ`, and `F (SETQ Y --)` be the same.

Note: There is some ambiguity in `(INSERT` *`EXPR`* `AFTER` *`FUNCTIONNAME`*`)`, as you might mean make *EXPR* be the function's first argument. Similarly, you cannot write `(REPLACE SETQ WITH SETQQ)` meaning change the name of the function. You must in these cases write `(INSERT` *`EXPR`* `AFTER` *`FUNCTIONNAME`* `1)`, and `(REPLACE SETQ 1 WITH SETQQ)`.

Occasionally, however, you may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as Interlisp attaches to atoms that appear as `CAR` of a list, versus those appearing elsewhere in a list. In general, you may not even *know* whether a particular atom is at the head of a list or not. Thus, when you write `(INSERT` *`EXPR`* `BEFORE FOO)`, you mean before the atom `FOO`, whether or not it is `CAR` of a list. By setting the variable `UPFINDFLG` to `NIL` (initially T), you can suppress the implicit UP that follows searches for atoms, and thus achieve the desired effect. With UPFINDFLG = NIL, following F FOO, for example, the current expression will be the atom FOO. In this case, the A, B, and : operations will operate with respect to the atom FOO. If you intend the operation to refer to the list which FOO heads, use the pattern (FOO --) instead .

Extract and Embed

Extraction involves replacing the current expression with one of its subexpressions (from any depth).

(XTR . @) [Editor Command]

Replaces the original current expression with the expression that is current after performing (LCL . @) (see warning about INSERT above). If the current expression after (LCL . @) is a *tail* of a higher expression, its first element is used.

If the extracted expression is a list, then after XTR has finished, the current expression will be that list. If the extracted expression is not a list, the new current expression will be a tail whose first element is that non-list.

For example, if the current expression is (COND ((NULL X) (PRINT Y))), (XTR PRINT), or (XTR 2 2) will replace the COND by the PRINT. The current expression after the XTR would be (PRINT Y).

If the current expression is (COND ((NULL X) Y) (T Z)), then (XTR Y) will replace the COND with Y, even though the current expression after performing (LCL Y) is ... Y). The current expression after the XTR would be ... Y followed by whatever followed the COND.

If the current expression *initially* is a tail, extraction works exactly the same as though the current expression were the first element in that tail. Thus if the current expression is ... (COND ((NULL X) (PRINT Y))) (RETURN Z)), then (XTR PRINT) will replace the COND by the PRINT, leaving (PRINT Y) as the current expression.

The extract command can also incorporate a location specification:

(EXTRACT $@_1$ FROM . $@_2$) [Editor Command]

Performs (LC . $@_2$) and then (XTR . $@_1$) (see warning about INSERT). The current edit chain is not changed, but UNFIND is set to the edit chain after the XTR was performed.

Note: $@_1$ is the *segment* between EXTRACT and FROM.

For example: If the current expression is (PRINT (COND ((NULL X) Y) (T Z))) then following (EXTRACT Y FROM COND), the current expression will be (PRINT Y). (EXTRACT 2 -1 FROM COND), (EXTRACT Y FROM 2), and (EXTRACT 2 -1 FROM 2) will all produce the same result.

While extracting replaces the current expression by a subexpression, embedding replaces the current expression with one containing *it* as a subexpression.

(MBD E_1 ... E_M) [Editor Command]

MBD substitutes the current expression for all instances of the atom & in E_1 ... E_M, and replaces the current expression with the result of that substitution. As with SUBST, a fresh copy is used for each substitution.

If & does not appear in E_1 ... E_M, the MBD is interpreted as (MBD (E_1 ... E_M &)).

MBD leaves the edit chain so that the larger expression is the new current expression.

Examples:

If the current expression is (PRINT Y), (MBD (COND ((NULL X) &) ((NULL (CAR Y)) & (GO LP)))) would replace (PRINT Y) with (COND ((NULL X) (PRINT Y)) ((NULL (CAR Y)) (PRINT Y) (GO LP))).

If the current expression is (RETURN X), (MBD (PRINT Y) (AND FLG &)) would replace it with the *two* expressions (PRINT Y) and (AND FLG (RETURN X)), i.e., if the (RETURN X) appeared in the cond clause (T (RETURN X)), after the MBD, the clause would be (T (PRINT Y) (AND FLG (RETURN X))).

If the current expression is (PRINT Y), then (MBD SETQ X) will replace it with (SETQ X (PRINT Y)). If the current expression is (PRINT Y), (MBD RETURN) will replace it with (RETURN (PRINT Y)).

If the current expression *initially* is a tail, embedding works exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (MBD SETQ X) would replace (PRINT Y) with (SETQ X (PRINT Y)).

The embed command can also incorporate a location specification:

(EMBED @ IN . *X*) [Editor Command]

(@ is the segment between EMBED and IN.) Does (LC . @) and then (MBD . *X*) (see warning about INSERT). Edit chain is not changed, but UNFIND is set to the edit chain after the MBD was performed.

Examples: (EMBED PRINT IN SETQ X), (EMBED 3 2 IN RETURN), (EMBED COND 3 1 IN (OR & (NULL X))).

WITH can be used for IN, and SURROUND can be used for EMBED, e.g., (SURROUND NUMBERP WITH (AND & (MINUSP X))).

EDITEMBEDTOKEN [Variable]

The special atom used in the MBD and EMBED commands is the value of this variable, initially &.

The MOVE Command

The MOVE command allows you to specify the expression to be moved, the place it is to be moved to, and the operation to be performed there, e.g., insert it before, insert it after, replace, etc.

(MOVE $@_1$ TO *COM* . $@_2$) [Editor Command]

($@_1$ is the segment between MOVE and TO.) *COM* is BEFORE, AFTER, or the name of a list command, e.g., :, N, etc. Performs (LC . $@_1$) (see warning about INSERT), and obtains the current expression there (or its first element, if it is a tail), which we will call *EXPR*; MOVE then goes back to the original edit chain, performs (LC . $@_2$) followed by (*COM EXPR*) (setting an internal flag so *EXPR* is not copied), then goes back to $@_1$ and deletes *EXPR*. The edit chain is not changed. UNFIND is set to the edit chain after (*COM EXPR*) was performed.

If $@_2$ specifies a location *inside of the expression to be moved*, a message is printed and an error is generated, e.g., (MOVE 2 TO AFTER X), where X is contained inside of the second element.

For example, if the current expression is (A B C D), (MOVE 2 TO AFTER 4) will make the new current expression be (A C D B). Note that 4 was executed as of the original edit chain, and that the second element had not yet been removed.

As the following examples taken from actual editing will show, the MOVE command is an extremely versatile and powerful feature of the editor.

```
*?
(PROG ((L L)) (EDLOC (CDDR C)) (RETURN (CAR L)))
*(MOVE 3 TO : CAR)
*?
(PROG ((L L)) (RETURN (EDLOC (CDDR C))))
*
*P
... (SELECTQ OBJPR & &) (RETURN &) LP2 (COND & &))
*(MOVE 2 TO N 1)
*P
... (SELECTQ OBJPR & & &) LP2 (COND & &))


*
*P
```

```
(OR (EQ X LASTAIL) (NOT &) (AND & & &))
*(MOVE 4 TO AFTER (BELOW COND))
*P
(OR (EQ X LASTAIL) (NOT &))
*\ P
... (& &) (AND & & &) (T & &))
*

*P
((NULL X) **COMMENT** (COND & &))
*(-3 (GO NXT]
*(MOVE 4 TO N (← PROG))
*P
((NULL X) **COMMENT** (GO NXT))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) (COND & &))
*(INSERT NXT BEFORE -1)
*P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) NXT (COND & &))
```

In the last example, you could have added the PROG label NXT and moved the COND in one operation by performing (MOVE 4 TO N (← PROG) (N NXT)). Similarly, in the next example, in the course of specifying $@_2$, the location where the expression was to be moved to, you also perform a structure modification, via (N (T)), thus creating the structure that will receive the expression being moved.

```
*P
((CDR &) **COMMENT** (SETQ CL &) (EDITSMASH CL & &))
*MOVE 4 TO N 0 (N (T)) -1]
*P
((CDR &) **COMMENT** (SETQ CL &))
*\ P
*(T (EDITSMASH CL & &))
*
```

If $@_2$ is NIL, or (HERE), the current position specifies where the operation is to take place. In this case, UNFIND is set to where the expression that was moved was originally located, i.e., $@_1$. For example:

```
*P
(TENEX)
*(MOVE ↑ F APPLY TO N HERE)
*P
(TENEX (APPLY & &))
*

*P
(PROG (& & & ATM IND VAL) (OR & &) **COMMENT** (OR & &)
(PRIN1 & T) (
PRIN1 & T) (SETQ IND     user typed Control-E

*(MOVE * TO BEFORE HERE)
*P
(PROG (& & & ATM IND VAL) (OR & &) (OR & &) (PRIN1 &
```

```
*P
(T (PRIN1 C-EXP T))
*(MOVE ↑ BF PRIN1 TO N HERE)
*P
(T (PRIN1 C-EXP T) (PRIN1 & T))
*
```

Finally, if $@_1$ is `NIL`, the `MOVE` command allows you to specify where the *current expression* is to be moved to. In this case, the edit chain is changed, and is the chain where the current expression was moved to; `UNFIND` is set to where it was.

```
*P
(SELECTQ OBJPR (&) (PROGN & &))
*(MOVE TO BEFORE LOOP)
*P
... (SELECTQ OBJPR & &) LOOP (FRPLACA DFPRP &) (FRPLACD DFPRP
&) (SELECTQ          user typed Control-E

*
```

Commands That Move Parentheses

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Of course, there will always be the same number of left parentheses as right parentheses in any list structure, since the parentheses are just a notational guide to the structure provided by `PRINT`. Thus, no command can insert or remove just one parenthesis, but this is suggestive of what actually happens.

In all six commands, *N* and *M* are used to specify an element of a list, usually of the current expression. In practice, *N* and *M* are usually positive or negative integers with the obvious interpretation. However, all six commands use the generalized `NTH` command `(NTH` *COM*`)` to find their element(s), so that *N*th element means the first element of the tail found by performing `(NTH` *N*`)`. In other words, if the current expression is `(LIST (CAR X) (SETQ Y (CONS W Z)))`, then `(BI 2 CONS)`, `(BI X -1)`, and `(BI X Z)` all specify the exact same operation.

All six commands generate an error if the element is not found, i.e., the `NTH` fails. All are undoable.

`(BI` *N M*`)` [Editor Command]

"Both In". Inserts a left parentheses before the *N*th element and after the *M*th element in the current expression. Generates an error if the *M*th element is not contained in the *N*th tail, i.e., the *M*th element must be "to the right" of the *N*th element.

Example: If the current expression is `(A B (C D E) F G)`, then `(BI 2 4)` will modify it to be `(A (B (C D E) F) G)`.

`(BI` *N*`)` [Editor Command]

Same as `(BI` *N N*`)`.

Example: If the current expression is `(A B (C D E) F G)`, then `(BI -2)` will modify it to be `(A B (C D E) (F) G)`.

`(BO` *N*`)` [Editor Command]

"Both Out". Removes both parentheses from the *N*th element. Generates an error if *N*th element is not a list.

Example: If the current expression is `(A B (C D E) F G)`, then `(BO D)` will modify it to be `(A B C D E F G)`.

`(LI` *N*`)` [Editor Command]

"Left In". Inserts a left parenthesis before the *N*th element (and a matching right parenthesis at the end of the current expression), i.e. equivalent to `(BI` *N* `-1)`.

Example: if the current expression is `(A B (C D E) F G)`, then `(LI 2)` will modify it to be `(A (B (C D E) F G))`.

`(LO` *N*`)` [Editor Command]

"Left Out". Removes a left parenthesis from the *N*th element. *All elements following the Nth element are deleted.* Generates an error if *N*th element is not a list.

Example: If the current expression is `(A B (C D E) F G)`, then `(LO 3)` will modify it to be `(A B C D E)`.

`(RI` *N M*`)` [Editor Command]

"Right In". Inserts a right parenthesis after the *M*th element of the *N*th element. The rest of the *N*th element is brought up to the level of the current expression.

Example: If the current expression is `(A (B C D E) F G)`, `(RI 2 2)` will modify it to be `(A (B C) D E F G)`. Another way of thinking about `RI` is to read it as "move the right parenthesis at the end of the *N*th element *in* to after its *N*th element."

`(RO` *N*`)` [Editor Command]

"Right Out". Removes the right parenthesis from the *N*th element, moving it to the end of the current expression. All elements following the *N*th element are moved inside of the *N*th element. Generates an error if *N*th element is not a list.

Example: If the current expression is (A B (C D E) F G), (RO 3) will modify it to be (A B (C D E F G)). Another way of thinking about RO is to read it as "move the right parenthesis at the end of the *N*th element *out* to the end of the current expression."

TO and THRU

EXTRACT, EMBED, DELETE, REPLACE, and MOVE can be made to operate on several contiguous elements, i.e., a segment of a list, by using in their respective location specifications the TO or THRU command.

($@_1$ THRU $@_2$) [Editor Command]

Does a (LC . $@_1$), followed by an UP, and then a (BI 1 $@_2$), thereby grouping the segment into a single element, and finally does a 1, making the final current expression be that element.

For example, if the current expression is (A (B (C D) (E) (F G H) I) J K), following (C THRU G), the current expression will be ((C D) (E) (F G H)).

($@_1$ TO $@_2$) [Editor Command]

Same as THRU except the last element not included, i.e., after the BI, an (RI 1 -2) is performed.

If both $@_1$ and $@_2$ are numbers, and $@_2$ is greater than $@_1$, then $@_2$ counts from the beginning of the current expression, the same as $@_1$. In other words, if the current expression is (A B C D E F G), (3 THRU 5) means (C THRU E) not (C THRU G). In this case, the corresponding BI command is (BI 1 $@_2$-$@_1$+1).

THRU and TO are not very useful commands by themselves; they are intended to be used in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor flag informing the above commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. Thus:

```
*P
         (PROG (& & ATM IND VAL WORD) (PRIN1 & T) (PRIN1 & T) (SETQ
         IND &)
         (SETQ VAL &) **COMMENT** (SETQQ      user typed Control-E


*(MOVE (3 THRU 4) TO BEFORE 7)
         *P
         (PROG (& & ATM IND VAL WORD) (SETQ IND &) (SETQ VAL &)
         (PRIN1 & T)
         (PRIN1 & T) **COMMENT**      user typed Control-E
```

```
*
          *P
          (* FAIL RETURN FROM EDITOR. USER SHOULD NOTE THE VALUES OF
          SOURCEXPR
          AND CURRENTFORM.  CURRENTFORM IS THE LAST FORM IN SOURCEXPR
          WHICH WILL
          HAVE BEEN TRANSLATED, AND IT CAUSED THE ERROR.)
          *(DELETE (USER THRU CURR$))
          =CURRENTFORM.
          *P
          (* FAIL RETURN FROM EDITOR.CURRENTFORM IS     user typed Control-E
```

```
*
          *P
          ... LP (SELECTO & & & & NIL) (SETQ Y &) OUT (SETQ FLG &)
          (RETURN Y))
          *(MOVE (1 TO OUT) TO N HERE]
          *P
          ... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & & & NIL)
          (SETQ Y &))
          *
```

```
*PP
          [PROG (RF TEMP1 TEMP2)
             (COND
                ((NOT (MEMB REMARG LISTING))
                   (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS))
          **COMMENT**
                   (SETQ TEMP2 (CADR TEMP1))
                   (GO SKIP))
                (T    **COMMENT**
                   (SETQ TEMP1 REMARG)))
             (NCONC1 LISTING REMARG)
             (COND
                ((NOT (SETQ TEMP2 (SASSOC
```

```
*(EXTRACT (SETQ THRU CADR) FROM COND)
          *P
          (PROG (RF TEMP1 TEMP2) (SETQ TEMP1 &) **COMMENT** (SETQ
          TEMP2 &) (NCONC1 LISTING REMARG) (COND & &     user typed Control-
          E

          *
```

TO and THRU can also be used directly with XTR, because XTR involves a location specification while A, B, :, and MBD do not. Thus in the previous example, if the current expression had been the COND, e.g.,you had first performed F COND, you could have used (XTR (SETQ THRU CADR)) to perform the extraction.

(@$_1$ TO) [Editor Command]

(@$_1$ THRU) [Editor Command]

Both are the same as (@$_1$ THRU -1), i.e., from @$_1$ through the end of the list.

Examples:

```
*P
(VALUE (RPLACA DEPRP &) (RPLACD &) (RPLACA VARSWORD
&) (RETURN))
*(MOVE (2 TO) TO N (← PROG))
*(N (GO VAR))
*P
(VALUE (GO VAR))
*P
(T **COMMENT** (COND &) **COMMENT** (EDITSMASH CL &
&) (COND &))
*(-3 (GO REPLACE))
*(MOVE (COND TO) TO N ↑ PROG (N REPLACE))
*P
(T **COMMENT** (GO REPLACE))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &)
DELETE (COND & &) REPLACE
(COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*

*PP
[LAMBDA (CLAUSALA X)
   (PROG (A D)
      (SETQ A CLAUSALA)
  LP  (COND
         ((NULL A)
            (RETURN)))
      (SERCH X A)
      (RUMARK (CDR A))
      (NOTICECL (CAR A))
      (SETQ A (CDR A))
      (GO LP]
*(EXTRACT (SERCH THRU NOT$) FROM PROG)
=NOTICECL
*P
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL
&))
*(EMBED (SERCH TO) IN (MAP CLAUSALA (FUNCTION (LAMBDA
(A) *]
*PP
[LAMBDA (CLAUSALA X)
   (MAP CLAUSALA
        (FUNCTION (LAMBDA (A)
                (SERCH X A)
                (RUMARK (CDR A))
                (NOTICECL (CAR A]
*
```

The R Command

`(R X Y)` [Editor Command]

Replaces all instances of *X* by *Y* in the current expression, e.g., `(R CAADR CADAR)`. Generates an error if there is not at least one instance.

The `R` command operates in conjunction with the search mechanism of the editor. The search proceeds as described in the Search Algorithm section above, and *X* can employ any of the patterns shown in the Commands That Search section above. Each time *X* matches an element of the structure, the element is replaced by (a copy of) *Y*; each time *X* matches a tail of the structure, the tail is replaced by (a copy of) *Y*.

For example, if the current expression is `(A (B C) (B . C))`,

`(R C D)` will change it to `(A (B D) (B . D))`,

`(R (... . C) D)` will change it to `(A (B C) (B . D))`,

`(R C (D E))` will change it to `(A (B (D E)) (B D E))`, and

`(R (... . NIL) D)` will change it to `(A (B C . D) (B . C) . D)`.

If *X* is an atom or string containing `$`s (escapes), `$`s appearing in *Y* stand for the characters matched by the corresponding `$` in *X*. For example, `(R FOO$ FIE$)` means for all atoms or strings that begin with `FOO`, replace the characters "`FOO`" by "`FIE`". Applied to the list `(FOO FOO2 XFOO1)`, *(R FOO$ FIE$)* would produce `(FIE FIE2 XFOO1)`, and `(R $FOO$ $FIE$)` would produce `(FIE FIE2 XFIE1)`. Similarly, `(R $D$ $A$)` will change `(LIST (CADR X) (CADDR Y))` to `(LIST (CAAR X) (CAADR))`. Note that `CADDR` was *not* changed to `CAAAR`, i.e., `(R $D$ $A$)` does not mean replace every `D` with `A`, but replace the first `D` in every atom or string by `A`. If you wanted to replace every `D` by `A`, you could perform `(LP (R $D$ $A$))`.

You will be informed of all such `$` replacements by a message of the form *X*`->`*Y*, e.g., `CADR->CAAR`.

If *X* matches a string, it will be replaced by a string. It does not matter whether *X* or *Y* themselves are strings, i.e. `(R $D$ $A$)`, `(R "$D$" $A$)`, `(R $D$ "$A$")`, and `(R "$D$" "$A$")` are equivalent. *X* will never match with a number, i.e., `(R $1 $2)` will not change 11 to 12.

The `$` (escape) feature can be used to delete or add characters, as well as replace them. For example, `(R $1 $)` will delete the terminating `1`’s from all literal atoms and strings.

Similarly, if an `$` in *X* does not have a mate in *Y*, the characters matched by the `$` are effectively deleted. For example, `(R $/$ $)` will change `AND/OR` to `AND`. There is no similar operation for changing `AND/OR` to `OR`, since the first `$` in *Y* always corresponds to the first `$` in *X*, the second **$** in *Y* to the second in *X*, etc. *Y* can also be a list containing `$`s, e.g., `(R $1 (CAR $))` will change `FOO1` to `(CAR FOO)`, `FIE1` to `(CAR FIE)`.

If *X* does not contain `$`s, `$` appearing in *Y* refers to the *entire* expression matched by *X*, e.g., `(R LONGATOM '$)` changes `LONGATOM` to `'LONGATOM`, `(R (SETQ X &) (PRINT $))` changes every `(SETQ X &)` to `(PRINT (SETQ X &))`. If *X* is a pattern containing an `$` pattern somewhere *within* it, the characters matched by the `$`s are not available, and for the purposes of replacement, the effect is the same as though *X* did not contain any `$`s. For example, if you type `(R (CAR F$) (PRINT $))`, the second `$` will refer to the entire expression matched by `(CAR F$)`.

Since `(R $X$ $Y$)` is a frequently used operation for **R**eplacing **C**haracters, the following command is provided:

`(RC X Y)` [Editor Command]

Equivalent to `(R $X$ $Y$)`

`R` and `RC` change all instances of *X* to *Y*. The commands `R1` and `RC1` are available for changing just one, (i.e., the first) instance of *X* to *Y*.

`(R1 X Y)` [Editor Command]

Find the first instance of *X* and replace it by *Y*.

`(RC1 X Y)` [Editor Command]

Equivalent to `(R1 $X$ $Y$)`.

In addition, while `R` and `RC` only operate within the current expression, `R1` and `RC1` will continue searching, a la the `F` command, until they find an instance of *x*, even if the search carries them beyond the current expression.

`(SW N M)` [Editor Command]

Switches the *N*th and *M*th elements of the current expression.

For example, if the current expression is `(LIST (CONS (CAR X) (CAR Y)) (CONS (CDR X) (CDR Y)))`, `(SW 2 3)` will modify it to be `(LIST (CONS (CDR X) (CDR Y)) (CONS (CAR X) (CAR Y)))`. The relative order of *N* and *M* is not important, i.e., `(SW 3 2)` and `(SW 2 3)` are equivalent.

`SW` uses the generalized `NTH` command `(NTH` *COM*`)` to find the *N*th and *M*th elements, a la the `BI-BO` commands.

Thus in the previous example, `(SW CAR CDR)` would produce the same result.

`(SWAP` $@_1$ $@_2$`)` [Editor Command]

Like `SW` except switches the expressions specified by $@_1$ and $@_2$, not the corresponding elements of the current expression, i.e. $@_1$ and $@_2$ can be at different levels in current expression, or one or both be outside of current expression.

Thus, using the previous example, `(SWAP CAR CDR)` would result in `(LIST (CONS (CDR X) (CAR Y)) (CONS (CAR X) (CDR Y)))`.

Commands That Print

`PP` [Editor Command]

Prettyprints the current expression.

`P` [Editor Command]

Prints the current expression as though PRINTLEVEL (Chapter 25) were set to 2.

`(P` *M*`)` [Editor Command]

Prints the *M*th element of the current expression as though PRINTLEVEL were set to 2.

`(P` *0*`)` [Editor Command]

Same as P.

`(P` *M N*`)` [Editor Command]

Prints the *M*th element of the current expression as though PRINTLEVEL were set to *N*.

`(P 0` *N*`)` [Editor Command]

Prints the current expression as though PRINTLEVEL were set to *N*.

`?` [Editor Command]

Same as (P 0 100).

Both (P *M*) and (P *M N*) use the generalized NTH command (NTH *COM*) to obtain the corresponding element, so that *M* does not have to be a number, e.g., (P COND 3) will work. PP causes all comments to be printed as **COMMENT** (see Chapter 26). P and ? print as **COMMENT** only those comments that are (top level) elements of the current expression. Lower expressions are not really seen by the editor; the printing command simply sets PRINTLEVEL and calls PRINT.

PP* [Editor Command]

Prettyprints current expression, *including* comments.

PP* is equivalent to PP except that it first resets **COMMENT**FLG to NIL (see Chapter 26).

PPV [Editor Command]

Prettyprints the current expression as a variable, i.e., no special treatment for LAMBDA, COND, SETQ, etc., or for CLISP.

PPT [Editor Command]

Prettyprints the current expression, printing CLISP translations, if any.

?= [Editor Command]

Prints the argument names and corresponding values for the current expression. Analagous to the ?= break command (Chapter 14). For example,

```
*P
(STRPOS "A0???" X N (QUOTE ?) T)
*?=
X = "A0???"
Y = X
START = N
SKIP = (QUOTE ?)
ANCHOR = T
TAIL =
```

The command MAKE (see below) is an imperative form of ?=. It allows you to specify a change to the element of the current expression that corresponds to a particular argument name.

All printing functions print to the terminal, regardless of the primary output file. All use the readtable T. No printing function ever changes the edit chain. All record the current edit chain for use by \P (above). All can be aborted with Control-E.

Commands for Leaving the Editor

`OK` [Editor Command]

Exits from the editor.

`STOP` [Editor Command]

Exits from the editor with an error. Mainly for use in conjunction with `TTY:` commands (see next section) that you want to abort.

Since all of the commands in the editor are errorset protected, you must exit from the editor via a command. `STOP` provides a way of distinguishing between a successful and unsuccessful (from your standpoint) editing session. For example, if you are executing `(MOVE 3 TO AFTER COND TTY:)`, and you exitsfrom the lower editor with an `OK`, the `MOVE` command will then complete its operation. If you want to abort the `MOVE` command, you must make the `TTY:` command generate an error. Do this by exiting from the lower editor with a `STOP` command. In this case, the higher editor's edit chain will not be changed by the `TTY:` command.

Actually, it is also possible to exit the editor by typing Control-D. `STOP` is preferred even if you are editing at the `EVALQT` level, as it will perform the necessary "wrapup" to insure that the changes made while editing will be undoable.

`SAVE` [Editor Command]

Exits from the editor and saves the "state of the edit" on the property list of the function or variable being edited under the property `EDIT-SAVE`. If the editor is called again on the same structure, the editing is effectively "continued," i.e., the edit chain, mark list, value of `UNFIND` and `UNDOLST` are restored.

For example:

```
*P
(NULL X)
*F COND P
(COND (& &) (T &))
*SAVE
FOO
← .
   .
   .
←EDITF(FOO)
EDIT
*P
(COND (& &) (T &))
*\ P
(NULL X)
*
```

SAVE is necessary only if you are editing many different expressions; an exit from the editor via OK always saves the state of the edit of that call to the editor on the property list of the atom EDIT, under the property name LASTVALUE. OK also remprops EDIT-SAVE from the property list of the function or variable being edited.

Whenever the editor is entered, it checks to see if it is editing the same expression as the last one edited. In this case, it restores the mark list and UNDOLST, and sets UNFIND to be the edit chain as of the previous exit from the editor. For example:

```
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
  .
  .
  .
*P
(COND & &)
*OK
FOO
← .
  .                  any number of LISPX inputs
  .                  except for calls to the editor
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
*\ P
(COND & &)
*
```

Furthermore, as a result of the history feature, if the editor is called on the same expression within a certain number of LISPX inputs (namely, the size of the history list, which can be changed with CHANGESLICE, Chapter 13) the state of the edit of that expression is restored, regardless of how many other expressions may have been edited in the meantime. For example:

```
←EDITF(FOO)
EDIT
*
  .
  .
  .
*P
(COND (& &) (& &) (&) (T &))
*OK
FOO
  .                  a small number of LISPX inputs,
  .                  including editing
  .
←EDITF(FOO)
EDIT
*\ P
(COND (& &) (& &) (&) (T &))
*
```

Thus you can always continue editing, including undoing changes from a previous editing session, if one of the following occurs:

1. No other expressions have been edited since that session (since saving takes place at *exit* time, intervening calls that were aborted via Control-D or exited via `STOP` will not affect the editor's memory).
2. That session was "sufficiently" recent.
3. It was ended with a `SAVE` command.

Nested Calls to Editor

`TTY:` [Editor Command]

Calls the editor recursively. You can then type in commands, and have them executed. The `TTY:` command is completed when you exit from the lower editor (see `OK` and `STOP` above).

The `TTY:` command is extremely useful. It enables you to set up a complex operation, and perform interactive attention-changing commands part way through it. For example, the command `(MOVE 3 TO AFTER COND 3 P TTY:)` allows you to interact, in effect, *within* the `MOVE` command. You can then verify for yourself that the correct location has been found, or complete the specification "by hand." In effect, `TTY:` says "I'll tell you what you should do when you get there."

The `TTY:` command operates by printing `TTY:` and then calling the editor. The initial edit chain in the lower editor is the one that existed in the higher editor at the time the `TTY:` command was entered. Until you exit from the lower editor, any attention changing commands you execute only affect the lower editor's edit chain. Of course, if you perform any structure modification commands while under a `TTY:` command, these will modify the structure in both editors, since it is the same structure. When the `TTY:` command finishes, the lower editor's edit chain becomes the edit chain of the higher editor.

`EF` [Editor Command]
`EV` [Editor Command]
`EP` [Editor Command]

Calls `EDITF` or `EDITV` or `EDITP` on `CAR` of current expression.

Manipulating the Characters of an Atom or String

RAISE [Editor Command]

An edit macro defined as UP followed by (I 1 (U-CASE (## 1))), i.e., it raises to uppercase the current expression, or if a tail, the first element of the current expression.

LOWER [Editor Command]

Similar to RAISE, except uses L-CASE.

CAP [Editor Command]

First does a RAISE, and then lowers all but the first character, i.e., the first character is left capitalized.

RAISE, LOWER, and CAP are all no-ops if the corresponding atom or string is already in that state.

(RAISE *X*) [Editor Command]

Equivalent to (I R (L-CASE *X*) *X*), i.e., changes every lowercase *X* to uppercase in the current expression.

(LOWER *X*) [Editor Command]

Similar to RAISE, except performs (I R *X* (L-CASE *X*)).

In both (RAISE *X*) and (LOWER *X*), *X* should be typed in uppercase.

REPACK [Editor Command]

Permits the "editing" of an atom or string.

REPACK operates by calling the editor recursively on UNPACK of the current expression, or if it is a list, on UNPACK of its first element. If the lower editor is exited successfully, i.e., via OK as opposed to STOP, the list of atoms is made into a single atom or string, which replaces the atom or string being "repacked." The new atom or string is always printed.

Example:

```
*P
... "THIS IS A LOGN STRING")
*REPACK
*EDIT
P
(T H I S % I S % A % L O G N % S T R I N G)
```

```
*(SW G N)
*OK
"THIS IS A LONG STRING"
*
```

This could also have been accomplished by `(R $GN$ $NG$)` or simply `(RC GN NG)`.

`(REPACK @)` [Editor Command]

Does `(LC . @)` followed by `REPACK`, e.g. `(REPACK THIS$)`.

Manipulating Predicates and Conditional Expressions

`JOINC` [Editor Command]

Used to join two neighboring `COND`s together, e.g. (COND $CLAUSE_1$ $CLAUSE_2$) followed by (COND $CLAUSE_3$ $CLAUSE_4$) becomes (COND $CLAUSE_1$ $CLAUSE_2$ $CLAUSE_3$ $CLAUSE_4$). `JOINC` does an `(F COND T)` first so that you don't have to be at the first `COND`.

`(SPLITC` *X*`)` [Editor Command]

Splits one `COND` into two. *X* specifies the last clause in the first `COND`, e.g. `(SPLITC 3)` splits (COND $CLAUSE_1$ $CLAUSE_2$ $CLAUSE_3$ $CLAUSE_4$) into (COND $CLAUSE_1$ $CLAUSE_2$) (COND $CLAUSE_3$ $CLAUSE_4$). Uses the generalized `NTH` command (NTH *COM*), so that *X* does not have to be a number, e.g., you can say `(SPLITC RETURN)`, meaning split after the clause containing `RETURN`. `SPLITC` also does an `(F COND T)` first.

`NEGATE` [Editor Command]

Negates the current expression, i.e. performs `(MBD NOT)`, except that is smart about simplifying. For example, if the current expression is: `(OR (NULL X) (LISTP X))`, `NEGATE` would change it to `(AND X (NLISTP X))`.

`NEGATE` is implemented via the function `NEGATE` (Chapter 3).

`SWAPC` [Editor Command]

Takes a conditional expression of the form `(COND (A B)(T C))` and rearranges it to an equivalent `(COND ((NOT A) C)(T B))`, or `(COND (A B) (C D))` to `(COND ((NOT A) (COND (C D))) (T B))`.

SWAPC is smart about negations (uses NEGATE) and simplifying CONDs. It always produces an equivalent expression. It is useful for those cases where one wants to insert extra clauses or tests.

History Commands in the Editor

All of your inputs to the editor are stored on the history list EDITHISTORY (see Chapter 13, the editor's history list, and all of the programmer's assistant commands for manipulating the history list, e.g. REDO, USE, FIX, NAME, etc., are available for use on events on EDITHISTORY. In addition, the following four history commands are recognized specially by the editor. They always operate on the last, i.e. most recent, event.

DO *COM* [Editor Command]

Allows you to supply the command name when it was omitted.

USE is useful when a command name is *incorrect*.

For example, suppose you want to perform (-2 (SETQ X (LIST Y Z))) but instead types just (SETQ X (LIST Y Z)). The editor will type SETQ ?, whereupon you can type DO -2. The effect is the same as though you had typed FIX, followed by (LI 1), (-1 -2), and OK, i.e., the command (-2 (SETQ X (LIST Y Z))) is executed. DO also works if the command is a line command.

!F [Editor Command]

Same as DO F.

In the case of !F, the previous command is always treated as though it were a line command, e.g., if you type (SETQ X &) and then !F, the effect is the same as though you had typed F (SETQ X &), not (F (SETQ X &)).

!E [Editor Command]

Same as DO E.

!N [Editor Command]

Same as DO N.

Miscellaneous Commands

NIL [Editor Command]

Unless preceded by F or BF, is always a no-op. Thus extra right parentheses or square brackets at the ends of commands are ignored.

CL [Editor Command]

Clispifies the current expression (see Chapter 21).

DW [Editor Command]

Dwimifies the current expression (see Chapter 21).

IFY [Editor Command]

If the current statement is a COND statement (Chapter 9), replaces it with an eqivalent IF statement.

GET* [Editor Command]

If the current expression is a comment pointer (see Chapter 26), reads in the full text of the comment, and replaces the current expression by it.

(* . *X*) [Editor Command]

X is the text of a comment. * ascends the edit chain looking for a "safe" place to insert the comment, e.g., in a COND clause, after a PROG statement, etc., and inserts (* . *X*) *after* that point, if possible, otherwise before. For example, if the current expression is (FACT (SUB1 N)) in

```
[COND
   ((ZEROP N) 1)
   (T (ITIMES N (FACT (SUB1 N]
```

then (* CALL FACT RECURSIVELY) would insert (* CALL FACT RECURSIVELY) *before* the ITIMES expression. If inserted after the ITIMES, the comment would then be (incorrectly) returned as the value of the COND. However, if the COND was itself a PROG statement, and hence its value was not being used, the comment could be (and would be) inserted after the ITIMES expression.

* does not change the edit chain, but UNFIND is set to where the comment was actually inserted.

GETD [Editor Command]

Essentially "expands" the current expression in line:

1. If (CAR of) the current expression is the name of a macro, expands the macro in line;
2. If a CLISP word, translates the current expression and replaces it with the translation;
3. If CAR is the name of a function for which the editor can obtain a symbolic definition, either in-core or from a file, substitutes the argument expressions for the corresponding argument names in the body of the definition and replaces the current expression with the result;
4. If CAR of the current expression is an open lambda, substitutes the arguments for the corresponding argument names in the body of the lambda, and then removes the lambda and argument list.

Warning: When expanding a function definition or open lambda expression, GETD does a simple substitution of the actual arguments for the formal arguments. Therefore, if any of the function arguments are used in other ways in the function definition (as functions, as record fields, etc.), they will simply be replaced with the actual arguments.

(MAKEFN (*FN* . *ACTUALARGS*) *ARGLIST* *N1* *N2*) [Editor Command]

The inverse of GETD: makes the current expression into a function. *FN* is the function name, *ARGLIST* its arguments. The argument names are substituted for the corresponding argument values in *ACTUALARGS*, and the result becomes the body of the function definition for *FN*. The current expression is then replaced with (*FN* . *ACTUALARGS*).

If N_1 and N_2 are supplied, (N_1 THRU N_2) is used rather than the current expression; if just N_1 is supplied, (N_1 THRU -1) is used.

If *ARGLIST* is omitted, MAKEFN will make up some arguments, using elements of *ACTUALARGS*, if they are literal atoms, otherwise arguments selected from (X Y Z A B C ...), avoiding duplicate argument names.

Example: If the current expression is (COND ((CAR X) (PRINT Y T)) (T (HELP))), then (MAKEFN (FOO (CAR X) Y) (A B)) will define FOO as (LAMBDA (A B) (COND (A (PRINT B T)) (T (HELP)))) and then replace the current expression with (FOO (CAR X) Y).

`(MAKE` *`ARGNAME EXP`*`)` [Editor Command]

Makes the value of *`ARGNAME`* be *`EXP`* in the call which is the current expression, i.e. a `?=` command following a `MAKE` will always print *`ARGNAME`* `=` *`EXP`*. For example:

```
*P
(JSYS)
*?=
JSYS[N;AC1,AC2,AC3,RESULTAC]
*(MAKE N 10)
*(MAKE RESULTAC 3)
*P
(JSYS 10 NIL NIL NIL 3)
```

`Q` [Editor Command]

Quotes the current expression, i.e. `MBD QUOTE`.

`D` [Editor Command]

Deletes the current expression, then prints new current expression, i.e. `(:) I P`.

Commands That Evaluate

`E` [Editor Command]

Causes the editor to call the Interlisp executive `LISPX` giving it the next input as argument. Example:

```
*E BREAK(FIE FUM)
(FIE FUM)
*E (FOO)

(FIE BROKEN)
:
```

`E` only works when when typed in, e.g, `(INSERT D BEFORE E)` will treat `E` as a pattern, and search for `E`.

`(E` *`X`*`)` [Editor Command]

Evaluates *X*, i.e., performs `(EVAL` *`X`*`)`, and prints the result on the terminal.

`(E` *`X`* `T)` [Editor Command]

Same as `(E` *`x`*`)` but does not print.

The `(E X)` and `(E X T)` commands are mainly intended for use by macros and subroutine calls to the editor; you would probably type in a form for evaluation using the more convenient format of the (atomic) `E` command.

`(I C X1 ... XN)` [Editor Command]

Executes the *editor command* `(C Y1 ... YN)` where `Yi = (EVAL Xi)`. If *C* is not an atom, *C* is evaluated also.

Examples:

`(I 3 (GETD 'FOO))` will replace the third element of the current expression with the definition of `FOO`.

`(I N FOO (CAR FIE))` will attach the value of `FOO` and `CAR` of the value of `FIE` to the end of the current expression.

`(I F = FOO T)` will search for an expression `EQ` to the value of `FOO`.

`(I (COND ((NULL FLG) '-1) (T 1)) FOO)`, if `FLG` is `NIL`, inserts the value of `FOO` before the first element of the current expression, otherwise replaces the first element by the value of `FOO`.

The `I` command sets an internal flag to indicate to the structure modification commands *not* to copy expression(s) when inserting, replacing, or attaching.

`EVAL` [Editor Command]

Does an `EVAL` of the current expression.

`EVAL`, line-feed, and the `GO` command together effectively allows you to "single-step" a program through its symbolic definition.

`GETVAL` [Editor Command]

Replaces the current expression by the result of evaluating it.

`(## COM1 COM2 ... COMN)` [NLambda NoSpread Function]

An nlambda, nospread function (not a command). Its value is what the current expression would be after executing the edit commands COM_1 ... COM_N starting from the present edit chain. Generates an error if any of COM_1 thru COM_N cause errors. The current edit chain is never changed.

Note: The A, B, :, INSERT, REPLACE, and CHANGE commands make special checks for ## forms in the expressions used for inserting or replacing, and use a copy of ## form instead (see the A,B, and : Commands section above). Thus, (INSERT (## 3 2) AFTER 1) is equivalent to (I INSERT (COPY (## 3 2)) 'AFTER 1).

Example: (I R 'X (## (CONS .. Z))) replaces all X's in the current expression by the first CONS containing a Z.

The I command is not very convenient for computing an *entire* edit command for execution, since it computes the command name and its arguments separately. Also, the I command cannot be used to compute an atomic command. The following two commands provide more general ways of computing commands.

(COMS X_1 ... X_M) [Editor Command]

Each X_i is evaluated and its value is executed as a command.

For example, (COMS (COND (X (LIST 1 X)))) will replace the first element of the current expression with the value of X if non-NIL, otherwise do nothing. The editor command NIL is a no-op (see the Miscellaneous Commands section above).

(COMSQ COM_1 ... COM_N) [Editor Command]

Executes COM_1 ... COM_N.

COMSQ is mainly useful in conjunction with the COMS command. For example, suppose you want to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the COMS command. You would then write (COMS (CONS 'COMSQ *X*)) where *X* computed the list of commands, e.g., (COMS (CONS 'COMSQ (GETP FOO 'COMMANDS))).

Commands That Test

(IF *X*) [Editor Command]

Generates an error *unless* the value of (EVAL *X*) is true. In other words, if (EVAL *X*) causes an error or (EVAL *X*) = NIL, IF will cause an error.

For some editor commands, the occurrence of an error has a well defined meaning, i.e., they use errors to branch on, as COND uses NIL and non-NIL. For example, an error condition in a location specification may simply mean "not this one, try the next." Thus the location specification `(IPLUS (E (OR (NUMBERP (## 3)) (ERROR!)) T))` specifies the first `IPLUS` whose second argument is a number. The `IF` command, by equating `NIL` to error, provides a more natural way of accomplishing the same result. Thus, an equivalent location specification is `(IPLUS (IF (NUMBERP (## 3))))`.

The `IF` command can also be used to select between two alternate lists of commands for execution.

`(IF` *X* *COMS*$_1$ *COMS*$_2$`)` [Editor Command]

If `(EVAL` *X*`)` is true, execute *COMS*$_1$; if `(EVAL` *X*`)` causes an error or is equal to `NIL`, execute *COMS*$_2$.

Thus `IF` is equivalent to

```
(COMS (CONS 'COMSQ
        (COND
           ((CAR (NLSETQ (EVAL X)))
              COMS1)
           (T   COMS2))))
```

For example, the command `(IF (READP T) NIL (P))` will print the current expression provided the input buffer is empty.

`(IF` *X* *COMS*$_1$`)` [Editor Command]

If `(EVAL` *X*`)` is true, execute *COMS*$_1$; otherwise generate an error.

`(LP` *COMS*$_1$ ... *COMS*$_N$`)` [Editor Command]

Repeatedly executes *COMS*$_1$ ... *COMS*$_N$ until an error occurs.

For example, `(LP F PRINT (N T))` will attach a `T` at the end of every `PRINT` expression. `(LP F PRINT (IF (## 3) NIL ((N T))))` will attach a `T` at the end of each print expression which does not already have a second argument. The form `(## 3)` will cause an error if the edit command `3` causes an error, thereby selecting `((N T))` as the list of commands to be executed. The `IF` could also be written as `(IF (CDDR (##)) NIL ((N T)))`.

When an error occurs, `LP` prints *N* `OCCURRENCES` where *N* is the number of times the commands were successfully executed. The edit chain is left as of the last complete successful execution of *COMS*$_1$ ... *COMS*$_N$.

(LPQ *COMS*$_1$ ... *COMS*$_N$) [Editor Command]

Same as LP but does not print the message *N* OCCURRENCES.

In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches MAXLOOP, initially set to 30. MAXLOOP can be set to NIL, which is equivalent to setting it to infinity. Since the edit chain is left as of the last successful completion of the loop, you can simply continue the LP command with REDO (see Chapter 13).

(SHOW *X*) [Editor Command]

X is a list of patterns. SHOW does a LPQ printing all instances of the indicated expression(s), e.g. (SHOW FOO (SETQ FIE &)) will print all FOOs and all (SETQ FIE &)s. Generates an error if there aren't any instances of the expression(s).

(EXAM *X*) [Editor Command]

Like SHOW except calls the editor recursively (via the TTY: command, see above) on each instance of the indicated espression(s) so that you can examine and/or change them.

(ORR *COMS*$_1$ ... *COMS*$_N$) [Editor Command]

ORR begins by executing *COMS*$_1$, a list of commands. If no error occurs, ORR is finished. Otherwise, ORR restores the edit chain to its original value, and continues by executing *COMS*$_2$, etc. If none of the command lists execute without errors, i.e., the ORR "drops off the end", ORR generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without an error.

NIL as a command list is perfectly legal, and will always execute successfully. Thus, making the last "argument" to ORR be NIL will insure that the ORR never causes an error. Any other atom is treated as (*ATOM*), i.e., the above example could be written as (ORR NX !NX NIL).

For example, (ORR (NX) (!NX) NIL) will perform a NX, if possible, otherwise a !NX, if possible, otherwise do nothing. Similarly, DELETE could be written as (ORR (UP (1)) (BK UP (2)) (UP (: NIL))).

Edit Macros

Many of the more sophisticated branching commands in the editor, such as ORR, IF, etc., are most often used in conjunction with edit macros. The macro feature permits you to define new commands and thereby expand the editor's repertoire, or redefine existing commands (to refer to the original definition of a built-in command when redefining it via a macro, use the `ORIGINAL` command, below).

Macros are defined by using the `M` command:

`(M` *C* *COMS*$_1$ `...` *COMS*$_N$`)` [Editor Command]

For *C* an atom, `M` defines *C* as an atomic command. If a macro is redefined, its new definition replaces its old. Executing *C* is then the same as executing the list of commands *COMS*$_1$ `...` *COMS*$_N$.

For example, `(M BP BK UP P)` will define `BP` as an atomic command which does three things, a `BK`, and `UP`, and a `P`. Macros can use commands defined by macros as well as built in commands in their definitions. For example, suppose `Z` is defined by `(M Z -1 (IF (READP T) NIL (P)))`, i.e., `Z` does a `-1`, and then if nothing has been typed, a `P`. Now we can define `ZZ` by `(M ZZ -1 Z)`, and `ZZZ` by `(M ZZZ -1 -1 Z) or (M ZZZ -1 ZZ)`.

Macros can also define list commands, i.e., commands that take arguments.

`(M (`*C*`) (`*ARG*$_1$ `...` *ARG*$_N$`)` *COMS*$_1$ `...` *COMS*$_M$`)` [Editor Command]

C an atom. `M` defines *C* as a list command. Executing `(`*C* *E*$_1$ `...` *E*$_N$`)` is then performed by substituting *E*$_1$ for *ARG*$_1$, `...` *E*$_N$ for *ARG*$_N$ throughout *COMS*$_1$ `...` *COMS*$_M$, and then executing *COMS*$_1$ `...` *COMS*$_M$.

For example, we could define a more general `BP` by `(M (BP) (N) (BK N) UP P)`. Thus, `(BP 3)` would perform `(BK 3)`, followed by an `UP`, followed by a `P`.

A list command can be defined via a macro so as to take a fixed or indefinite number of "arguments", as with spread vs. nospread functions. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the "argument list" is *atomic*, the command takes an indefinite number of arguments.

`(M (`*C*`)` *ARG* *COMS*$_1$ `...` *COMS*$_M$`)` [Editor Command]

If *C*, *ARG* are both atoms, this defines *C* as a list command. Executing `(`*C* *E*$_1$ `...` *E*$_N$`)` is performed by substituting `(`*E*$_1$ `...` *E*$_N$`)`, i.e., `CDR` of the command, for *ARG* throughout *COMS*$_1$ `...` *COMS*$_M$, and then executing *COMS*$_1$ `...` *COMS*$_M$.

For example, the command `2ND` (see the Location Specification section above), could be defined as a macro by `(M (2ND) X (ORR ((LC . X) (LC . X))))`.

For all editor commands, "built in" commands as well as commands defined by macros as atomic commands and list definitions are *completely* independent. In other words, the existence of an atomic definition for *C* in *no* way affects the treatment of *C* when it appears as CAR of a list command, and the existence of a list definition for *C* in *no* way affects the treatment of *C* when it appears as an atom. In particular, *C* can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used.

Once *C* is defined as an atomic command via a macro definition, it will *not* be searched for when used in a location specification, unless it is preceded by an F. Thus (INSERT -- BEFORE BP) would not search for BP, but instead perform a BK, and UP, and a P, and then do the insertion. The corresponding also holds true for list commands.

Occasionally, your will want to employ the S command in a macro to save some temporary result. For example, the SW command could be defined as:

```
(M (SW) (N M)
    (NTH N)
    (S FOO 1)
    MARK
    0
    (NTH M)
    (S FIE 1)
    (I 1 FOO)
    ←←
    (I 1 FIE))
```

Since this version of SW sets FOO and FIE, using SW may have undesirable side effects, especially when the editor was called from deep in a computation, we would have to be careful to make up unique names for dummy variables used in edit macros, which is bothersome. Furthermore, it would be impossible to define a command that called itself recursively while setting free variables. The BIND command solves both problems.

(BIND $COMS_1$ ... $COMS_N$) [Editor Command]

Binds three dummy variables #1, #2, #3, (initialized to NIL), and then executes the edit commands $COMS_1$ ... $COMS_N$. BIND uses a PROG to make these bindings, so they are only in effect while the commands are being executed and BINDs can be used recursively; the variables #1, #2, and #3 will be rebound each time BIND is invoked.

Thus, we can write SW safely as:

```
(M (SW) (N M)
    (BIND (NTH N)
          (S #1 1)
          MARK
          0
          (NTH M)
          (S #2 1)
          (I 1 #1)
```

←←   (I 1 #2)))

(ORIGINAL *COMS*$_1$ ... *COMS*$_N$)   [Editor Command]

Executes *COMS*$_1$ ... *COMS*$_N$ without regard to macro definitions. Useful for redefining a built in command in terms of itself., i.e. effectively allows you to "advise" edit commands.

User macros are stored on a list USERMACROS. The file package command USERMACROS (Chapter 17) is available for dumping all or selected user macros.

Undo

Each command that causes structure modification automatically adds an entry to the front of UNDOLST that contains the information required to restore all pointers that were changed by that command.

UNDO   [Editor Command]

Undoes the last, i.e., most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., MBD undone. The edit chain is then *exactly* what it was before the "undone" command had been performed. If there are no commands to undo, UNDO types nothing saved.

!UNDO   [Editor Command]

Undoes all modifications performed during this editing session, i.e. this call to the editor. As each command is undone, its name is printed a la UNDO. If there is nothing to be undone, !UNDO prints nothing saved.

Undoing an event containing an I, E, or S command will also undo the side effects of the evaluation(s), e.g., undoing (I 3 (/NCONC FOO FIE)) will not only restore the third element but also restore FOO. Similarly, undoing an S command will undo the set. See the discussion of UNDO in Chapter 13. (If the I command was typed directly to the editor, /NCONC would automatically be substituted for NCONC as described in Chapter 13.)

Since UNDO and !UNDO cause structure modification, they also add an entry to UNDOLST. However, UNDO and !UNDO entries are skipped by UNDO, e.g., if you perform an INSERT, and then an MBD, the first UNDO will undo the MBD, and the second will undo the INSERT. However, you can also specify precisely which commands you want undone by identifying the corresponding entry. In this case, you can undo an UNDO command, e.g., by typing UNDO UNDO, or undo a !UNDO command, or undo a command other than that most recently performed.

Whenever you *continue* an editing session, the undo information of the previous session is protected by inserting a special blip, called an undo-block, on the front of UNDOLST. This undo-block will terminate the operation of a !UNDO, thereby confining its effect to the current session, and will similarly prevent an UNDO command from operating on commands executed in the previous session.

Thus, if you enter the editor continuing a session, and immediately execute an UNDO or !UNDO, the editor will type BLOCKED instead of NOTHING SAVED. Similarly, if you execute several commands and then undo them all, another UNDO or !UNDO will also cause BLOCKED to be typed.

UNBLOCK [Editor Command]

Removes an undo-block. If executed at a non-blocked state, i.e., if UNDO or !UNDO *could* operate, types NOT BLOCKED.

TEST [Editor Command]

Adds an undo-block at the front of UNDOLST.

Note that TEST together with !UNDO provide a "tentative" mode for editing, i.e., you can perform a number of changes, and then undo all of them with a single !UNDO command.

(UNDO *EventSpec*) [Editor Command]

EventSpec is an event specification (see Chapter 13). Undoes the indicated event on the history list. In this case, the event does not have to be in the current editing session, even if the previous session has not been unblocked as described above. However, you do have to be editing the same expression as was being edited in the indicated event.

If the expressions differ, the editor types the warning message "different expression," and does not undo the event. The editor enforces this to avoid your accidentally undoing a random command by giving the wrong event specification.

EDITDEFAULT

Whenever a command is not recognized, i.e., is not "built in" or defined as a macro, the editor calls an internal function, EDITDEFAULT, to determine what action to take. Since EDITDEFAULT is part of the edit block, you cannot advise or redefine it as a means of augmenting or extending the editor. However, you can accomplish this via EDITUSERFN. If the value of the variable EDITUSERFN is T, EDITDEFAULT calls the function EDITUSERFN giving it the command as an argument. If EDITUSERFN returns a non-NIL value, its value is interpreted as a single command and executed. Otherwise, the error correction procedure described below is performed.

If a location specification is being executed, an internal flag informs EDITDEFAULT to treat the command as though it had been preceded by an F.

If the command is a list, an attempt is made to perform spelling correction on the CAR of the command (unless DWIMFLG = NIL) using EDITCOMSL, a list of all list edit commands. If spelling correction is successful, the correct command name is RPLACAed into the command, and the editor continues by executing the command. In other words, if you type (LP F PRINT (MBBD AND (NULL FLG))), only one spelling correction will be necessary to change MBBD to MBD. If spelling correction is not successful, an error is generated.

Note: When a macro is defined via the M command, the command name is added to EDITCOMSA or EDITCOMSL, depending on whether it is an atomic or list command. The USERMACROS file package command is aware of this, and provides for restoring EDITCOMSA and EDITCOMSL.

If the command is atomic, the procedure followed is a little more elaborate.

1. If the command is one of the list commands, i.e., a member of EDITCOMSL, and there is additional input on the same terminal line, treat the entire line as a single list command. The line is read using READLINE (see Chapter 13), so the line can be terminated by a square bracket, or by a carriage return not preceded by a space. You may omit parentheses for any list command typed in at the top level (provided the command is not also an atomic command, e.g. NX, BK). For example,

```
*P
          (COND (& &) (T &))
          *XTR 3 2]
          *MOVE TO AFTER LP
          *
```

If the command is on the list EDITCOMSL but no additional input is on the terminal line, an error is generated. For example:

```
*P
          (COND (& &) (T &))
          *MOVE

          MOVE ?
          *
```

If the command is on EDITCOMSL, and *not* typed in directly, e.g., it appears as one of the commands in a LP command, the procedure is similar, with the rest of the command stream at that level being treated as "the terminal line", e.g. (LP F (COND (T &)) XTR 2 2).

If the command is being executed in location context, EDITDEFAULT does not get this far, e.g., (MOVE TO AFTER COND XTR 3) will search for XTR, *not* execute it. However, (MOVE TO AFTER COND (XTR 3)) will work.

2. If the command was typed in and the first character in the command is an 8, treat the 8 as a mistyped left parenthesis, and and the rest of the line as the arguments to the command, e.g.,

```
*P
          (COND (& &) (T &))
          *8-2 (Y (RETURN Z)))
          =(-2
          *P
          (COND (Y &) (& &) (T &))
```

3. If the command was typed in, is the name of a function, and is followed by NIL or a list CAR of which is not an edit command, assume you forgot to type E and intend to apply the function to its arguments, type =E and the function name, and perform the indicated computation, e.g.

```
*BREAK(FOO)
          =E BREAK
          (FOO)
          *
```

4. If the last character in the command is P, and the first *N*-1 characters comprise a number, assume that you intended two commands, e.g.,

```
*P
          (COND (& &) (T &))
          *0P
          =0 P
          (SETQ X (COND & &))
```

5. Attempt spelling correction using EDITCOMSA, and if successful, execute the corrected command.

6. If there is additional input on the same line, or command stream, spelling correct using EDITCOMSL as a spelling list, e.g.,

```
*MBBD SETQ X
          =MBD
          *
```

7. Otherwise, generate an error.

Time Stamps

Whenever a function is edited, and changes were made, the function is time-stamped (by EDITE), which consists of inserting a comment of the form (* *USERS-INITIALS DATE*). *USERS-INITIALS* is the value of the variable INITIALS. After greeting (see Chapter 12), the function SETINITIALS is called. SETINITIALS searches INITIALSLST, a list of elements of the form (*USERNAME* . *INITIALS*) or (*USERNAME FIRSTNAME INITIALS*). If your name is found, INITIALS is set accordingly. If your username name is *not* found on INITIALSLST, INITIALS is set to the value of DEFAULTINITIALS, initially edited:. Thus, the default is to always time stamp. To suppress time stamping, you must either include an entry of the form (*USERNAME*) on INITIALSLST, or set DEFAULTINITIALS to NIL before greeting, i.e. in your user profile, or else, *after* greeting, explicitly set INITIALS to NIL.

If you want your functions to be time stamped with your initials when edited, include a file package command command of the form (ADDVARS (INITIALSLST (*USERNAME* . *INITIALS*))) in your INIT.LISP file (see Chapter 12).

The following three functions may be of use for specialized applications with respect to time-stamping: (FIXEDITDATE *EXPR*) which, given a lambda expression, inserts or smashes a time-stamp comment; (EDITDATE? *COMMENT*) which returns T if *COMMENT* is a time stamp; and (EDITDATE *OLDATE INITLS*) which returns a new time-stamp comment. If *OLDATE* is a time-stamp comment, it will be reused.

Warning with Declarations

CAUTION: There is a feature of the BYTECOMPILER that is not supported by SEdit or the XCL compiler. It is possible to insert a comment at the beginning of your function that looks like

(* DECLARATIONS: --)

The tail, or -- section, of this comment is taken as a set of local record declarations which are then used by the compiler in that function just as if they had been declared globally. See the "Compiler" section in Chapter 3 of these Notes for additional behavior in XCL.

SEdit does not recognize such declarations. Thus, if the "Expand" command is used, the expansion will not be done with these record declarations in effect. The code that you see in SEdit will not be the same code compiled by the BYTECOMPILER.

[This page intentionally left blank]

17. FILE MANAGER

Warning: The subsystem within Medley used for managing collections of definitions (of functions, variables, etc.) is known as the "File Manager." This terminology is confusing, because the word "file" is also used in the more conventional sense as meaning a collection of data stored on some physical media. Unfortunately, it is not possible to change this terminology at this time, because many functions and variables (`MAKEFILE`*,* `FILEPKGTYPES`*, etc.) incorporate the word "file" in their names.*

Most implementations of Lisp treat symbolic files as unstructured text, much as they are treated in most conventional programming environments. Function definitions are edited with a character-oriented text editor, and then the changed definitions (or sometimes the entire file) is read or compiled to install those changes in the running memory image. Interlisp incorporates a different philosophy. A symbolic file is considered as a database of information about a group of data objects---function definitions, variable values, record declarations, etc. The text in a symbolic file is never edited directly. Definitions are edited only after their textual representations on files have been converted to data-structures that reside inside the Lisp address space. The programs for editing definitions inside Medley can therefore make use of the full set of data-manipulation capabilities that the environment already provides, and editing operations can be easily intermixed with the processes of evaluation and compilation.

Medley is thus a "resident" programming environment, and as such it provides facilities for moving definitions back and forth between memory and the external databases on symbolic files, and for doing the bookkeeping involved when definitions on many symbolic files with compiled counterparts are being manipulated. The file manager provides those capabilities. It shoulders the burden of keeping track of where things are and what things have changed so that you don’t have to. The file manager also keeps track of which files have been modified and need to be updated and recompiled.

The file manager is integrated into many other system packages. For example, if only the compiled version of a file is loaded and you attempt to edit a function, the file manager will attempt to load the source of that function from the appropriate symbolic file. In many cases, if a datum is needed by some program, the file manager will automatically retrieve it from a file if it is not already in your working environment.

Some of the operations of the file manager are rather complex. For example, the same function may appear in several different files, or the symbolic or compiled files may be in different directories, etc. Therefore, this chapter does not document how the file manager works in each and every situation, but instead makes the deliberately vague statement that it does the "right" thing with respect to keeping track of what has been changed, and what file operations need to be performed in accordance with those changes.

For a simple illustration of what the file manager does, suppose that the symbolic file `FOO` contains the functions `FOO1` and `FOO2`, and that the file `BAR` contains the functions `BAR1` and `BAR2`. These two files could be loaded into the environment with the function `LOAD`:

```
← (LOAD 'FOO)
FILE CREATED  4-MAR-83 09:26:55
```

```
FOOCOMS
{DSK}FOO.;1
← (LOAD 'BAR)
FILE CREATED  4-MAR-83 09:27:24
BARCOMS
{DSK}BAR.;1
```

Now, suppose that we change the definition of FOO2 with the editor, and we define two new functions, NEW1 and NEW2. At that point, the file manager knows that the in-memory definition of FOO2 is no longer consistent with the definition in the file FOO, and that the new functions have been defined but have not yet been associated with a symbolic file and saved on permanent storage. The function FILES? summarizes this state of affairs and enters into an interactive dialog in which we can specify what files the new functions are to belong to.

```
← (FILES?)
FOO...to be dumped.
    plus the functions: NEW1,NEW2
want to say where the above go ? Yes
(functions)
NEW1  File name: BAR
NEW2  File name: ZAP
  new file ? Yes
NIL
```

The file manager knows that the file FOO has been changed, and needs to be dumped back to permanent storage. This can be done with MAKEFILE.

```
←(MAKEFILE 'FOO)
{DSK}FOO.;2
```

Since we added NEW1 to the old file BAR and established a new file ZAP to contain NEW2, both BAR and ZAP now also need to be dumped. This is confirmed by a second call to FILES?:

```
←(FILES?)
BAR, ZAP...to be dumped.
FOO...to be listed.
FOO...to be compiled
NIL
```

We are also informed that the new version we made of FOO needs to be listed (sent to a printer) and that the functions on the file must be compiled.

Rather than doing several MAKEFILEs to dump the files BAR and ZAP, we can simply call CLEANUP. Without any further user interaction, this will dump any files whose definitions have been modified. CLEANUP will also send any unlisted files to the printer and recompile any files which need to be recompiled. CLEANUP is a useful function to use at the end of a debugging session. It will call FILES? if any new objects have been defined, so you do not lose the opportunity to say explicitly where those belong. In effect, the function CLEANUP executes all the operations necessary to make the your permanent files consistent with the definitions in the current core-image.

```
← (CLEANUP)
FOO...compiling {DSK}FOO.;2
```

```
    .
    .
    .
BAR...compiling {DSK}BAR.;2
    .
    .
    .
ZAP...compiling {DSK}ZAP.;1
    .
    .
    .
```

In addition to the definitions of functions, symbolic files in Interlisp can contain definitions of a variety of other types, e.g. variable values, property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file manager uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a symbol), a definition of a given type (called the file manager type). Note that the same name may have several definitions of different types. For example, a symbol may have both a function definition and a variable definition. The file manager also keeps track of the files that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

Symbolic files on permanent storage devices are referred to by names that obey the naming conventions of those devices, usually including host, directory, and version fields. When such definition groups are noticed by the file manager, they are assigned simple *root names* and these are used by all file manager operations to refer to those groups of definitions. The root name for a group is computed from its full permanent storage name by applying the function ROOTFILENAME; this strips off the host, directory, version, etc., and returns just the simple name field of the file. For each file, the file manager also has a data structure that describes what definitions it contains. This is known as the commands of the file, or its "filecoms". By convention, the filecoms of a file whose root name is *X* is stored as the value of the symbol *X*COMS. For example, the value of FOOCOMS is the filecoms for the file FOO. This variable can be directly manipulated, but the file manager contains facilities such as FILES? which make constructing and updating filecoms easier, and in some cases automatic. See the Functions for Manipulating File Command Lists section.

The file manager is able to maintain its databases of information because it is notified by various other routines in the system when events take place that may change that database. A file is "noticed" when it is loaded, or when a new file is stored (though there are ways to explicitly notice files without completely loading all their definitions). Once a file is noticed, the file manager takes it into account when modifying filecoms, dumping files, etc. The file manager also needs to know what typed definitions have been changed or what new definitions have been introduced, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file manager operations (LOAD, TCOMPL, PRETTYDEF, etc.), as well as those functions that define or change data, (EDITF, EDITV, EDITP, DWIM corrections to user functions) interact with the file manager. Also, *typed-in* assignment of variables or property values is noticed by the file manager. (Note that modifications to variable or property values during the execution of a function body are not noticed.) In some cases the marking procedure can be subtle, e.g. if you edit a property list using EDITP, only those properties whose values are actually changed (or added) are marked.

All file manager operations can be disabled with FILEPKGFLG.

FILEPKGFLG [Variable]

The file manager can be disabled by setting FILEPKGFLG to NIL. This will turn off noticing files and marking changes. FILEPKGFLG is initially T.

The rest of this chapter goes into further detail about the file manager. Functions for loading and storing symbolic files are presented first, followed by functions for adding and removing typed definitions from files, moving typed definitions from one file to another, determining which file a particular definition is stored in, and so on.

Loading Files

The functions below load information from symbolic files into the Interlisp environment. A symbolic file contains a sequence of Interlisp expressions that can be evaluated to establish specified typed definitions. The expressions on symbolic files are read using FILERDTBL as the read table.

The loading functions all have an argument *LDFLG*. *LDFLG* affects the operation of DEFINE, DEFINEQ, RPAQ, RPAQ?, and RPAQQ. While a source file is being loaded, DFNFLG (Chapter 10) is rebound to *LDFLG*. Thus, if *LDFLG* = NIL, and a function is redefined, a message is printed and the old definition saved. If *LDFLG* = T , the old definition is simply overwritten. If *LDFLG* = PROP, the functions are stored as "saved" definitions on the property lists under the property EXPR instead of being installed as the active definitions. If *LDFLG* = ALLPROP, not only function definitions but also variables set by RPAQQ, RPAQ, RPAQ? are stored on property lists (except when the variable has the value NOBIND, in which case they are set to the indicated value regardless of DFNFLG).

Another option is available for loading systems for others to use and who wish to suppress the saving of information used to aid in development and debugging. If *LDFLG* = SYSLOAD, LOAD will:

1. Rebind `DFNFLG` to `T`, so old definitions are simply overwritten

2. Rebind `LISPXHIST` to `NIL`, thereby making the `LOAD` not be undoable and eliminating the cost of saving undo information (Chapter 13)

3. Rebind `ADDSPELLFLG` to `NIL`, to suppress adding to spelling lists

4. Rebind `FILEPKGFLG` to `NIL`, to prevent the file from being "noticed" by the file manager

5. Rebind `BUILDMAPFLG` to `NIL`, to prevent a file map from being constructed

6. After the load has completed, set the filecoms variable and any filevars variables to `NOBIND`

7. Add the file name to `SYSFILES` rather than `FILELST`

A filevars variable is any variable appearing in a file manager command of the form `(`*FILECOM* `*` *VARIABLE*`)` (see the FileVars section). Therefore, if the filecoms includes `(FNS * FOOFNS)`, `FOOFNS` is set to `NOBIND`. If you want the value of such a variable to be retained, even when the file is loaded with *LDFLG* = `SYSLOAD`, then you should replace the variable with an equivalent, *non-atomic* expression, such as `(FNS * (PROGN FOOFNS))`.

All functions that have *LDFLG* as an argument perform spelling correction using `LOADOPTIONS` as a spelling list when *LDFLG* is not a member of `LOADOPTIONS`. `LOADOPTIONS` is initially `(NIL T PROP ALLPROP SYSLOAD)`.

(**LOAD** *FILE LDFLG PRINTFLG*) [Function]

Reads successive expressions from *FILE* (with `FILERDTBL` as read table) and evaluates each as it is read, until it reads either `NIL`, or the single atom `STOP`. Note that `LOAD` can be used to load both symbolic and compiled files. Returns *FILE* (full name).

If *PRINTFLG* = `T`, `LOAD` prints the value of each expression; otherwise it does not.

(**LOAD?** *FILE LDFLG PRINTFLG*) [Function]

Similar to `LOAD` except that it does not load *FILE* if it has already been loaded, in which case it returns `NIL`.

`LOAD?` loads *FILE* except when the *same* version of the file has been loaded (either from the same place, or from a copy of it from a different place). Specifically, `LOAD?` considers that *FILE* has already been loaded if the full name of *FILE* is on `LOADEDFILELST` (see the Noticing Files section) or the date stored on the `FILEDATES` property of the root file name of *FILE* is the same as the `FILECREATED` expression on *FILE*.

(**LOADFNS** *FNS FILE LDFLG VARS*) [Function]

Permits selective loading of definitions. *FNS* is a list of function names, a single function name, or `T`, meaning to load all of the functions on the file. *FILE* can be either a compiled or symbolic file. If a compiled definition is loaded, so are all compiler-generated subfunctions. The interpretation of *LDFLG* is the same as for LOAD.

If *FILE* = NIL, LOADFNS will use WHEREIS (see the Storing Files section) to determine where the first function in *FNS* resides, and load from that file. Note that the file must previously have been "noticed". If WHEREIS returns NIL, and the WHEREIS library package has been loaded, LOADFNS will use the WHEREIS data base to find the file containing *FN*.

VARS specifies which non-DEFINEQ expressions are to be loaded (i.e., evaluated). It is interpreted as follows:

T Means to load all non-DEFINEQ expressions.

NIL Means to load none of the non-DEFINEQ expressions.

VARS Means to evaluate all variable assignment expressions (beginning with RPAQ, RPAQQ, or RPAQ?, see the Functions Used Within Source Files section).

Any other symbol Means the same as specifying a list containing that atom.

A list If *VARS* is a list that is not a valid function definition, each element in *VARS* is "matched" against each non-DEFINEQ expression, and if any elements in *VARS* "match" successfully, the expression is evaluated. "Matching" is defined as follows: If an element of *VARS* is an atom, it matches an expression if it is EQ to either the CAR or the CADR of the expression. If an element of *VARS* is a list, it is treated as an edit pattern (see Chapter 16), and matched with the entire expression (using EDIT4E, described in Chapter 16). For example, if *VARS* was (FOOCOMS DECLARE: (DEFLIST & (QUOTE MACRO))), this would cause (RPAQQ FOOCOMS ...), all DECLARE:s, and all DEFLISTs which set up MACROs to be read and evaluated.

A function definition If *VARS* is a list and a valid function definition ((FNTYP *VARS*) is true), then LOADFNS will invoke that function on every non-DEFINEQ expression being considered, applying it to two arguments, the first and second elements in the expression. If the function returns NIL, the expression will be skipped; if it returns a non-NIL symbol (e.g., T), the expression will be evaluated; and if it returns a list, this list is evaluated instead of the expression. The file pointer is set to the very beginning of the expression before calling the *VARS* function definition, so it may read the entire expression if necessary. If the function returns a symbol, the file pointer is reset and the expression is READ or SKREAD. However, the file pointer is not reset when the function returns a list, so the function must leave it set immediately after the expression that it has presumably read.

LOADFNS returns a list of:

1. The names of the functions that were found
2. A list of those functions not found (if any) headed by the symbol NOT-FOUND:
3. All of the expressions that were evaluated
4. A list of those members of *VARS* for which no corresponding expressions were found (if any), again headed by the symbol NOT-FOUND:

For example:

```
← (LOADFNS '(FOO FIE FUM) FILE NIL '(BAZ (DEFLIST &)))
(FOO FIE (NOT-FOUND: FUM) (RPAQ BAZ ...) (NOT-FOUND:
(DEFLIST &)))
```

(**LOADVARS** *VARS FILE LDFLG*) [Function]

Same as (LOADFNS NIL *FILE LDFLG VARS*).

(**LOADFROM** *FILE FNS LDFLG*) [Function]

Same as (LOADFNS *FNS FILE LDFLG* T).

Once the file manager has noticed a file, you can edit functions contained in the file without explicitly loading them. Similarly, those functions which have not been modified do not have to be loaded in order to write out an updated version of the file. Files are normally noticed (i.e., their contents become known to the file manager) when either the symbolic or compiled versions of the file are loaded. If the file is *not* going to be loaded completely, the preferred way to notice it is with LOADFROM. You can also load some functions at the same time by giving LOADFROM a second argument, but it is normally used simply to inform the file manager about the existence and contents of a particular file.

(**LOADBLOCK** *FN FILE LDFLG*) [Function]

Calls LOADFNS on those functions contained in the block declaration containing *FN* (see Chapter 18). LOADBLOCK is designed primarily for use with symbolic files, to load the EXPRs for a given block. It will not load a function which already has an in-core EXPR definition, and it will not load the block name, unless it is also one of the block functions.

(**LOADCOMP** *FILE LDFLG*) [Function]

Performs all operations on *FILE* associated with compilation, i.e. evaluates all expressions under a DECLARE: EVAL@COMPILE, and "notices" the function and variable names by adding them to the lists NOFIXFNSLST and NOFIXVARSLST (see Chapter 21).

Thus, if building a system composed of many files with compilation information scattered among them, all that is required to compile one file is to LOADCOMP the others.

(**LOADCOMP?** *FILE LDFLG*) [Function]

Similar to LOADCOMP, except it does not load if file has already been loaded (with LOADCOMP), in which case its value is NIL.

LOADCOMP? will load the file even if it has been loaded with LOAD, LOADFNS, etc. The only time it will not load the file is if the file has already been loaded with LOADCOMP.

FILESLOAD provides an easy way for you to load a series of files, setting various options:

(**FILESLOAD** $FILE_1$ . . . $FILE_N$) [NLambda NoSpread Function]

Loads the files $FILE_1$ . . . $FILE_N$ (all arguments unevaluated). If any of these arguments are lists, they specify certain loading options for all following files (unless changed by another list). Within these lists, the following commands are recognized:

FROM *DIR* Search the specified directories for the file. *DIR* can either be a single directory, or a list of directories to search in order. For example, (FILESLOAD (FROM {ERIS}<LISPCORE>SOURCES>) ...) will search the directory {ERIS}<LISPCORE>SOURCES> for the files. If this is not specified, the default is to search the contents of DIRECTORIES (see Chapter 24).

If FROM is followed by the key word VALUEOF, the following word is evaluated, and the value is used as the list of directories to search. For example, (FILESLOAD (FROM VALUEOF FOO) ...) will search the directory list that is the value of the variable FOO.

As a special case, if *DIR* is a symbol, and the symbol *DIR*DIRECTORIES is bound, the value of this variable is used as the directory search list. For example, since the variable LISPUSERSDIRECTORIES (see Chapter 24) is commonly used to contain a list of directories containing "library" packages, (FILESLOAD (FROM LISPUSERS) ...) can be used instead of (FILESLOAD (FROM VALUEOF LISPUSERSDIRECTORIES) ...)

If a FILESLOAD is read and evaluated while loading a file, and it doesn't contain a FROM expression, the default is to search the directory containing the FILESLOAD expression before the value of DIRECTORIES. FILESLOAD expressions can be dumped on files using the FILES file manager command.

SOURCE Load the source version of the file rather than the compiled version.

COMPILED Load the compiled version of the file.

If COMPILED is specified, the compiled version will be loaded, if it is found. The source will not be loaded. If neither SOURCE or COMPILED is specified, the compiled version of the file will be loaded if it is found, otherwise the source will be loaded if it is found.

LOAD Load the file by calling LOAD, if it has not already been loaded. This is the default unless LOADCOMP or LOADFROM is specified.

If LOAD is specified, FILESLOAD considers that the file has already been loaded if the root name of the file has a non-NIL FILEDATES property. This is a somewhat different algorithm than LOAD? uses. In particular, FILESLOAD will not load a newer version of a file that has already been loaded.

LOADCOMP Load the file with LOADCOMP? rather than LOAD. Automatically implies SOURCE.

LOADFROM Load the file with LOADFROM rather than LOAD.

NIL, T, PROP
ALLPROP
SYSLOAD The loading function is called with its *LDFLG* argument set to the specified token. *LDFLG* affects the operation of the loading functions by resetting DFNFLG (see Chapter 10) to *LDFLG* during the loading. If none of these tokens are specified, the value of the variable LDFLG is used if it is bound, otherwise NIL is used.

NOERROR If NOERROR is specified, no error occurs when a file is not found.

Each list determines how all further files in the lists are loaded, unless changed by another list. The tokens above can be joined together in a single list. For example,

```
(FILESLOAD (LOADCOMP) NET (SYSLOAD FROM VALUEOF
NEWDIRECTORIES) CJSYS)
```

will call LOADCOMP? to load the file NET searching the value of DIRECTORIES, and then call LOADCOMP? to load the file CJSYS with *LDFLG* set to SYSLOAD, searching the directory list that is the value of the variable NEWDIRECTORIES.

FILESLOAD expressions can be dumped on files using the FILES file manager command.

Storing Files

(**MAKEFILE** *FILE OPTIONS REPRINTFNS SOURCEFILE*) [Function]

Makes a new version of the file *FILE*, storing the information specified by *FILE*'s filecoms. Notices *FILE* if not previously noticed. Then, it adds *FILE* to NOTLISTEDFILES and NOTCOMPILEDFILES.

OPTIONS is a symbol or list of symbols which specify options. By specifying certain options, MAKEFILE can automatically compile or list *FILE*. Note that if *FILE* does not contain any function definitions, it is not compiled even when *OPTIONS* specifies C or RC. The options are spelling corrected using the list MAKEFILEOPTIONS. If spelling correction fails, MAKEFILE generates an error. The options are interpreted as follows:

C
RC After making *FILE*, MAKEFILE will compile *FILE* by calling TCOMPL (if C is specified) or RECOMPILE (if RC is specified). If there are any block declarations specified in the filecoms for *FILE*, BCOMPL or BRECOMPILE will be called instead.

If F, ST, STF, or S is the *next* item on *OPTIONS* following C or RC, it is given to the compiler as the answer to the compiler's question LISTING? (see Chapter 18). For example, (MAKEFILE 'FOO '(C F LIST)) will dump FOO, then TCOMPL or BCOMPL it specifying that functions are not to be redefined, and finally list the file.

LIST After making *FILE*, MAKEFILE calls LISTFILES to print a hardcopy listing of *FILE*.

CLISPIFY MAKEFILE calls PRETTYDEF with CLISPIFYPRETTYFLG = T (see Chapter 21). This causes CLISPIFY to be called on each function defined as an EXPR before it is prettyprinted.

Alternatively, if *FILE* has the property FILETYPE with value CLISP or a list containing CLISP, PRETTYDEF is called with CLISPIFYPRETTYFLG reset to CHANGES, which will cause CLISPIFY to be called on all functions marked as having been changed. If *FILE* has property FILETYPE with value CLISP, the compiler will DWIMIFY its functions before compiling them (see Chapter 18).

FAST MAKEFILE calls PRETTYDEF with PRETTYFLG = NIL (see Chapter 26). This causes data objects to be printed rather than prettyprinted, which is much faster.

REMAKE MAKEFILE "remakes" *FILE*: The prettyprinted definitions of functions that have not changed are copied from an earlier version of the symbolic file. Only those functions that have changed are prettyprinted.

NEW MAKEFILE does *not* remake *FILE*. If MAKEFILEREMAKEFLG = T (the initial setting), the default for all calls to MAKEFILE is to remake. The NEW option can be used to override this default.

REPRINTFNS and *SOURCEFILE* are used when remaking a file.

FILE is not added to NOTLISTEDFILES if *FILE* has on its property list the property FILETYPE with value DON'TLIST, or a list containing DON'TLIST. *FILE* is not added to NOTCOMPILEDFILES if *FILE* has on its property list the property FILETYPE with value DON'TCOMPILE, or a list containing DON'TCOMPILE. Also, if *FILE* does not contain any function definitions, it is not added to NOTCOMPILEDFILES, and it is not compiled even when *OPTIONS* specifies C or RC.

If a remake is *not* being performed, MAKEFILE checks the state of *FILE* to make sure that the entire source file was actually LOADed. If *FILE* was loaded as a compiled file, MAKEFILE prints the message CAN'T DUMP: ONLY THE COMPILED FILE HAS BEEN LOADED. Similarly, if only some of the symbolic definitions were loaded via LOADFNS or LOADFROM, MAKEFILE prints CAN'T DUMP: ONLY SOME OF ITS SYMBOLICS HAVE BEEN LOADED. In both cases, MAKEFILE will then ask you if it should dump anyway; if you decline, MAKEFILE does not call PRETTYDEF, but simply returns (*FILE* NOT DUMPED) as its value.

You can indicate that *FILE* must be block compiled together with other files as a unit by putting a list of those files on the property list of each file under the property FILEGROUP. If *FILE* has a FILEGROUP property, the compiler will not be called until all files on this property have been dumped that need to be.

MAKEFILE operates by rebinding PRETTYFLG, PRETTYTRANFLG, and CLISPIFYPRETTYFLG, evaluating each expression on MAKEFILEFORMS (under errorset protection), and then calling PRETTYDEF.

PRETTYDEF calls PRETTYPRINT with its second argument *PRETTYDEFLG* = T, so whenever PRETTYPRINT (and hence MAKEFILE) start printing a new function, the name of that function is printed if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

(**MAKEFILES** *OPTIONS FILES*) [Function]

Performs (MAKEFILE *FILE OPTIONS*) for each file on *FILES* that needs to be dumped. If *FILES* = NIL, FILELST is used. For example, (MAKEFILES 'LIST) will make and list all files that have been changed. In this case, if any typed definitions for any items have been defined or changed and they are *not* contained in one of the files on FILELST, MAKEFILES calls ADDTOFILES? to allow you to specify where these go. MAKEFILES returns a list of all files that are made.

(**CLEANUP** $FILE_1$ $FILE_2$ ... $FILE_N$) [NLambda NoSpread Function]

Dumps, lists, and recompiles (with RECOMPILE or BRECOMPILE) any of the specified files (unevaluated) requiring the corresponding operation. If no files are specified, FILELST is used. CLEANUP returns NIL.

CLEANUP uses the value of the variable CLEANUPOPTIONS as the *OPTIONS* argument to MAKEFILE. CLEANUPOPTIONS is initially (RC), to indicate that the files should be recompiled. If CLEANUPOPTIONS is set to (RC F), no listing will be performed, and no functions will be redefined as the result of compiling. Alternatively, if $FILE_1$ is a list, it will be interpreted as the list of options regardless of the value of CLEANUPOPTIONS.

(**FILES?**) [Function]

Prints on the terminal the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled, plus the names of any functions and other typed definitions (if any) that are not contained in any file. If there are any, FILES? then calls ADDTOFILES? to allow you to specify where these go.

(**ADDTOFILES?** —) [Function]

Called from MAKEFILES, CLEANUP, and FILES? when there are typed definitions that have been marked as changed which do not belong to any file. ADDTOFILES? lists the names of the changed items, and asks if you want to specify where these items should be put. If you answer N(o), ADDTOFILES? returns NIL without taking any action. If you answer], this is taken to be an answer to each question that would be asked, and all the changed items are marked as dummy items to be ignored. Otherwise, ADDTOFILES? prints the name of each changed item, and accepts one of the following responses:

A file name
A filevar

If you give a file name or a variable whose value is a list (a filevar), the item is added to the corresponding file or list, using ADDTOFILE.

If your response is not the name of a file on FILELST or a variable whose value is a list, you will be asked whether it is a new file. If you say no, then ADDTOFILES? will check whether the item is the name of a list, i.e., whether its value is a list. If not, youwill be asked whether it is a new list.

line-feed — Same as your previous response.

space
carriage return — Take no action.

] — The item is marked as a dummy item by adding it to NILCOMS. This tells the file manager simply to ignore this item.

[— The "definition" of the item in question is prettyprinted to the terminal, and then you are asked again about its disposition.

(— ADDTOFILES? prompts with "LISTNAME: (", you type in the name of a list, i.e. a variable whose value is a list, terminated by a). The item will then only be added to (under) a command in which the named list appears as a filevar. If none are found, a message is printed, and you are asked again. For example, you define a new function FOO3. When asked where it goes, you type (FOOFNS). If the command (FNS * FOOFNS) is found, FOO3 will be added to the value of FOOFNS. If instead you type (FOOCOMS), and the command (COMS * FOOCOMS) is found, then FOO3 will be added to a command for dumping functions that is contained in FOOCOMS.

If the named list is not also the name of a file, you can simply type it in without parenthesis as described above.

@ — ADDTOFILES? prompts with "Near: (", you type in the name of an object, and the item is then inserted in a command for dumping objects (of its type) that contains the indicated name. The item is inserted immediately after the indicated name.

(**LISTFILES** $FILE_1$ $FILE_2$ ... $FILE_N$) [NLambda NoSpread Function]

Lists each of the specified files (unevaluated). If no files are given, NOTLISTEDFILES is used. Each file listed is removed from NOTLISTEDFILES if the listing is completed. For each file not found, LISTFILES prints the message *FILENAME* NOT FOUND and proceeds to the next file.

LISTFILES calls the function LISTFILES1 on each file to be listed. Normally, LISTFILES1 is defined to simply call SEND.FILE.TO.PRINTER (see Chapter 29), but you can advise or redefine LISTFILES1 for more specialized applications.

Any lists inside the argument list to LISTFILES are interpreted as property lists that set the various printing options, such as the printer, number of copies, banner page name, etc (see see Chapter 29). Later properties override earlier ones. For example,

```
(LISTFILES FOO (HOST JEDI) FUM (#COPIES 3) FIE)
```

will cause one copy of FOO to be printed on the default printer, and one copy of FUM and three copies of FIE to be printed on the printer JEDI.

(**COMPILEFILES** *FILE*$_1$ *FILE*$_2$ ... *FILE*$_N$) [NLambda NoSpread Function]

Executes the RC and C options of MAKEFILE for each of the specified files (unevaluated). If no files are given, NOTCOMPILEDFILES is used. Each file compiled is removed from NOTCOMPILEDFILES. If *FILE*$_1$ is a list, it is interpreted as the *OPTIONS* argument to MAKEFILES. This feature can be used to supply an answer to the compiler's LISTING? question, e.g., (COMPILEFILES (STF)) will compile each file on NOTCOMPILEDFILES so that the functions are redefined without the EXPRs definitions being saved.

(**WHEREIS** *NAME TYPE FILES FN*) [Function]

TYPE is a file manager type. WHEREIS sweeps through all the files on the list *FILES* and returns a list of all files containing *NAME* as a *TYPE*. WHEREIS knows about and expands all file manager commands and file manager macros. *TYPE* = NIL defaults to FNS (to retrieve function definitions). If *FILES* is not a list, the value of FILELST is used.

If *FN* is given, it should be a function (with arguments *NAME*, *FILE*, and *TYPE*) which is applied for every file in *FILES* that contains *NAME* as a *TYPE*. In this case, WHEREIS returns NIL.

If the WHEREIS library package has been loaded, WHEREIS is redefined so that *FILES* = T means to use the whereis package data base, so WHEREIS will find *NAME* even if the file has not been loaded or noticed. *FILES* = NIL always means use FILELST.

Remaking a Symbolic File

Most of the time that a symbolic file is written using MAKEFILE, only a few of the functions that it contains have been changed since the last time the file was written. Rather than prettprinting all of the functions, it is often considerably faster to "remake" the file, copying the prettprinted definitions of unchanged functions from an earlier version of the symbolic file, and only prettyprinting those functions that have been changed.

MAKEFILE will remake the symbolic file if the REMAKE option is specified. If the NEW option is given, the file is not remade, and all of the functions are prettprinted. The default action is specified by the value of MAKEFILEREMAKEFLG: if T (its initial value), MAKEFILE will remake files unless the NEW option is given; if NIL, MAKEFILE will not remake unless the REMAKE option is given.

Note: If the file has never been loaded or dumped, for example if the filecoms were simply set up in memory, then MAKEFILE will never attempt to remake the file, regardless of the setting of MAKEFILEREMAKEFLG, or whether the REMAKE option was specified.

When MAKEFILE is remaking a symbolic file, you can explicitly indicate the functions which are to be prettyprinted and the file to be used for copying the rest of the function definitions from via the *REPRINTFNS* and *SOURCEFILE* arguments to MAKEFILE. Normally, both of these arguments are defaulted to NIL. In this case, *REPRINTFNS* will be set to those functions that have been changed since the last version of the file was written. For *SOURCEFILE*, MAKEFILE obtains the full name of the most recent version of the file (that it knows about) from the FILEDATES property of the file, and checks to make sure that the file still exists and has the same file date as that stored on the FILEDATES property. If it does, MAKEFILE uses that file as *SOURCEFILE*. This procedure permits you to LOAD or LOADFROM a file in a different directory, and still be able to remake the file with MAKEFILE. In the case where the most recent version of the file cannot be found, MAKEFILE will attempt to remake using the *original* version of the file (i.e., the one first loaded), specifying as *REPRINTFNS* the union of all changes that have been made since the file was first loaded, which is obtained from the FILECHANGES property of the file. If both of these fail, MAKEFILE prints the message "CAN'T FIND EITHER THE PREVIOUS VERSION OR THE ORIGINAL VERSION OF *FILE*, SO IT WILL HAVE TO BE WRITTEN ANEW", and does not remake the file, i.e. will prettyprint all of the functions.

When a remake is specified, MAKEFILE also checks to see how the file was originally loaded. If the file was originally loaded as a compiled file, MAKEFILE will call LOADVARS to obtain those DECLARE: expressions that are contained on the symbolic file, but not the compiled file, and hence have not been loaded. If the file was loaded by LOADFNS (but not LOADFROM), then LOADVARS is called to obtain any non-DEFINEQ expressions. Before calling LOADVARS to re-load definitions, MAKEFILE asks you, e.g. "Only the compiled version of FOO was loaded, do you want to LOADVARS the (DECLARE: .. DONTCOPY ..) expressions from {DSK}<MYDIR>FOO.;3?". You can respond Yes to execute the LOADVARS and continue the MAKEFILE, No to proceed with the MAKEFILE without performing the LOADVARS, or Abort to abort the MAKEFILE. You may wish to skip the LOADVARS if you had circumvented the file manager in some way, and loading the old definitions would overwrite new ones.

Remaking a symbolic file is considerably faster if the earlier version has a *file map* indicating where the function definitions are located (see the File Maps section), but it does not depend on this information.

Loading Files in a Distributed Environment

Each Interlisp source and compiled code file contains the full filename of the file, including the host and directory names, in a FILECREATED expression at the beginning of the file. The compiled code file also contains the full file name of the source file it was created from. In earlier versions of Interlisp, the file manager used this information to locate the appropriate source file when "remaking" or recompiling a file.

This turned out to be a bad feature in distributed environments, where users frequently move files from one place to another, or where files are stored on removable media. For example, suppose you MAKEFILE to a floppy, and then copy the file to a file server. If you loaded and edited the file from a file server, and tried to do MAKEFILE, it would try to locate the source file on the floppy, which is probably no longer loaded.

Currently, the file manager searches for sources file on the connected directory, and on the directory search path (on the variable DIRECTORIES). If it is not found, the host/directory information from the FILECREATED expression be used.

Warning: One situation where the new algorithm does the wrong thing is if you explicitly LOADFROM a file that is not on your directory search path. Future MAKEFILEs and CLEANUPs will search the connected directory and DIRECTORIES to find the source file, rather than using the file that the LOADFROM was done from. Even if the correct file is on the directory search path, you could still create a bad file if there is another version of the file in an earlier directory on the search path. In general, you should either explicitly specify the *SOURCEFILE* argument to MAKEFILE to tell it where to get the old source, or connect to the directory where the correct source file is.

Marking Changes

The file manager needs to know what typed definitions have been changed, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file manager operations (LOAD, TCOMPL, PRETTYDEF, etc.), as well as those functions that define or change data, (EDITF, EDITV, EDITP, DWIM corrections to user functions) interact with the file manager by marking changes. Also, *typed-in* assignment of variables or property values is noticed by the file manager. (If a program modifies a variable or property value, this is not noticed.) In some cases the marking procedure can be subtle, e.g. if you edit a property list using EDITP, only those properties whose values are actually changed (or added) are marked.

The various system functions which create or modify objects call MARKASCHANGED to mark the object as changed. For example, when a function is defined via DEFINE or DEFINEQ, or modified via EDITF, or a DWIM correction, the function is marked as being a changed object of type FNS. Similarly, whenever a new record is declared, or an existing record redeclared or edited, it is marked as being a changed object of type RECORDS, and so on for all of the other file manager types.

You can also call MARKASCHANGED directly to mark objects of a particular file manager type as changed:

(**MARKASCHANGED** *NAME TYPE REASON*) [Function]

Marks *NAME* of type *TYPE* as being changed. MARKASCHANGED returns *NAME*. MARKASCHANGED is undoable.

REASON is a symbol that indicated how *NAME* was changed. MARKASCHANGED recognizes the following values for *REASON*:

DEFINED Used to indicate the creation of *NAME*, e.g. from DEFINEQ (Chapter 10).

CHANGED Used to indicate a change to *NAME*, e.g. from the editor.

DELETED Used to indicate the deletion of *NAME*, e.g. by DELDEF.

CLISP Used to indicate the modification of *NAME* by CLISP translation.

For backwards compatibility, MARKASCHANGED also accepts a *REASON* of T (=DEFINED) and NIL (=CHANGED). New programs should avoid using these values.

The variable MARKASCHANGEDFNS is a list of functions that MARKASCHANGED calls (with arguments *NAME*, *TYPE*, and *REASON*). Functions can be added to this list to "advise" MARKASCHANGED to do additional work for all types of objects. The WHENCHANGED file manager type property (see the Defining New File Manager Types section) can be used to specify additional actions when MARKASCHANGED gets called on specific types of objects.

(**UNMARKASCHANGED** *NAME TYPE*) [Function]

Unmarks *NAME* of type *TYPE* as being changed. Returns *NAME* if *NAME* was marked as changed and is now unmarked, NIL otherwise. UNMARKASCHANGED is undoable.

(**FILEPKGCHANGES** *TYPE LST*) [NoSpread Function]

If *LST* is not specified (as opposed to being NIL), returns a list of those objects of type *TYPE* that have been marked as changed but not yet associated with their corresponding files (see the File Manager Types section). If *LST* is specified, FILEPKGCHANGES sets the corresponding list. (FILEPKGCHANGES) returns a list of *all* objects marked as changed as a list of elements of the form (*TYPENAME* . *CHANGEDOBJECTS*).

Some properties (e.g. EXPR, ADVICE, MACRO, I.S.OPR, etc.) are used to implement other file manager types. For example, if you change the value of the property I.S.OPR, you are really changing an object of type I.S.OPR. The effect is the same as though you had redefined the i.s.opr via a direct call to the function I.S.OPR. If a property whose value has been changed or added does not correspond to a specific file manager type, then it is marked as a changed object of type PROPS whose *name* is (*VARIABLENAME PROPNAME*) (except if the property name has a property PROPTYPE with value IGNORE).

Similarly, if you change a variable which implements the file manager type ALISTS (as indicated by the appearance of the property VARTYPE with value ALIST on the variable's property list), only those entries that are actually changed are marked as being changed objects of type ALISTS. The "name" of the object will be (*VARIABLENAME KEY*) where *KEY* is CAR of the entry on the alist that is being marked. If the variable corresponds to a specific file manager type other than ALISTS, e.g., USERMACROS, LISPXMACROS, etc., then an object of that type is marked. In this case, the name of the changed object will be CAR of the corresponding entry on the alist. For example, if you edit LISPXMACROS and change a definition for PL, then the object PL of type LISPXMACROS is marked as being changed.

Noticing Files

Already existing files are "noticed" by LOAD or LOADFROM (or by LOADFNS or LOADVARS when the *VARS* argument is T. New files are noticed when they are constructed by MAKEFILE, or when definitions are first associated with them via FILES? or ADDTOFILES?. Noticing a file updates certain lists and properties so that the file manager functions know to include the file in their operations. For example, CLEANUP will only dump files that have been noticed.

You can explicitly tell the file manager to notice a newly-created file by defining the filecoms for the file, and calling ADDFILE:

(**ADDFILE** *FILE*) [Function]

Tells the file manager that *FILE* should be recognized as a file; it adds *FILE* to FILELST, and also sets up the FILE property of *FILE* to reflect the current set of changes which are "registered against" *FILE*.

The file manager uses information stored on the property list of the root name of noticed files. The following property names are used:

FILE [Property Name]

When a file is noticed, the property FILE, value ((*FILECOMS* . *LOADTYPE*)) is added to the property list of its root name. *FILECOMS* is the variable containing the filecoms of the file. *LOADTYPE* indicates *how* the file was loaded, e.g., completely loaded, only partially loaded as with LOADFNS, loaded as a compiled file, etc.

The property FILE is used to determine whether or not the corresponding file has been modified since the last time it was loaded or dumped. CDR of the FILE property records by type those items that have been changed since the last MAKEFILE. Whenever a file is dumped, these items are moved to the property FILECHANGES, and CDR of the FILE property is reset to NIL.

FILECHANGES [Property Name]

The property FILECHANGES contains a list of all changed items since the file was loaded (there may have been several sequences of editing and rewriting the file). When a file is dumped, the changes in CDR of the FILE property are added to the FILECHANGES property.

FILEDATES [Property Name]

The property FILEDATES contains a list of version numbers and corresponding file dates for this file. These version numbers and dates are used for various integrity checks in connection with *re*making a file.

FILEMAP [Property Name]

The property FILEMAP is used to store the filemap for the file. This is used to directly load individual functions from the middle of a file.

To compute the root name, ROOTFILENAME is applied to the name of the file as indicated in the FILECREATED expression appearing at the front of the file, since this name corresponds to the name the file was originally made under. The file manager detects that the file being noticed is a compiled file (regardless of its name), by the appearance of more than one FILECREATED expressions. In this case, each of the files mentioned in the following FILECREATED expressions are noticed. For example, if you perform (BCOMPL '(FOO FIE)), and subsequently loads FOO.DCOM, both FOO and FIE will be noticed.

When a file is noticed, its root name is added to the list FILELST:

FILELST [Variable]

Contains a list of the root names of the files that have been noticed.

LOADEDFILELST [Variable]

Contains a list of the actual names of the files as loaded by LOAD, LOADFNS, etc. For example, if you perform (LOAD '<NEWLISP>EDITA.COM;3), EDITA will be added to FILELST, but <NEWLISP>EDITA.COM;3 is added to LOADEDFILELST. LOADEDFILELST is not used by the file manager; it is maintained solely for your benefit.

Distributing Change Information

Periodically, the function UPDATEFILES is called to find which file(s) contain the elements that have been changed. UPDATEFILES is called by FILES?, CLEANUP, and MAKEFILES, i.e., any procedure that requires the FILE property to be up to date. This procedure is followed rather than updating the FILE property after each change because scanning FILELST and examining each file manager command can be a time-consuming process; this is not so noticeable when performed in conjunction with a large operation like loading or writing a file.

UPDATEFILES operates by scanning FILELST and interrogating the file manager commands for each file. When (if) any files are found that contain the corresponding typed definition, the name of the element is added to the value of the property FILE for the corresponding file. Thus, after UPDATEFILES has completed operating, the files that need to be dumped are simply those files on FILELST for which CDR of their FILE property is non-NIL. For example, if you load the file FOO containing definitions for FOO1, FOO2, and FOO3, edit FOO2, and then call UPDATEFILES, (GETPROP 'FOO 'FILE) will be ((FOOCOMS . T) (FNS FOO2)). If any objects marked as changed have not been transferred to the FILE property for some file, e.g., you define a new function but forget (or declines) to add it to the file manager commands for the corresponding file, then both FILES? and

CLEANUP will print warning messages, and then call ADDTOFILES? to permit you to specify on which files these items belong.

You can also invoke UPDATEFILES directly:

(**UPDATEFILES** — —) [Function]

(UPDATEFILES) will update the FILE properties of the noticed files.

File Manager Types

In addition to the definitions of functions and values of variables, source files in Interlisp can contain a variety of other information, e.g. property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file manager uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a symbol), a definition of a given type (called the file manager type). Note that the same name may have several definitions of different types. For example, a symbol may have both a function definition and a variable definition. The file manager also keeps track of the file that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

A file manager type is an abstract notion of a class of objects which share the property that every object of the same file manager type is stored, retrieved, edited, copied etc., by the file manager in the same way. Each file manager type is identified by a symbol, which can be given as an argument to the functions that manipulate typed definitions. You may define new file manager types, as described in the Defining New Package Types section.

FILEPKGTYPES [Variable]

The value of FILEPKGTYPES is a list of all file manager types, including any that you may have defined.

The file manager is initialized with the following built-in file manager types:

ADVICE [File Manager Type]

Used to access "advice" modifying a function (see Chapter 15).

ALISTS [File Manager Type]

Used to access objects stored on an association list that is the value of a symbol (see Chapter 3).

A variable is declared to have an association list as its value by putting on its property list the property VARTYPE with value ALIST. In this case, each dotted pair on the list is an object of type ALISTS. When the value of such a variable is changed, only those entries in the association list that are actually changed or added are marked as changed objects of type ALISTS (with "name" (SYMBOL *KEY*)). Objects of type ALISTS are dumped via the ALISTS or ADDVARS file manager commands.

Note that some association lists are used to "implement" other file manager types. For example, the value of the global variable USERMACROS implements the file manager type USERMACROS and the values of LISPXMACROS and LISPXHISTORYMACROS implement the file manager type LISPXMACROS. This is indicated by putting on the property list of the variable the property VARTYPE with value a list of the form (ALIST *FILEPKGTYPE*). For example, (GETPROP 'LISPXHISTORYMACROS 'VARTYPE) => (ALIST LISPXMACROS).

COURIERPROGRAMS [File Manager Type]

Used to access Courier programs (see Chapter 31).

EXPRESSIONS [File Manager Type]

Used to access lisp expressions that are put on a file by using the REMEMBER programmers assistant command (Chapter 13), or by explicitly putting the P file manager command on the filecoms.

FIELDS [File Manager Type]

Used to access fields of records. The "definition" of an object of type FIELDS is a list of all the record declarations which contain the name. See Chapter 8.

FILEPKGCOMS [File Manager Type]

Used to access file manager commands and types. A single name can be defined both as a file manager type and a file manager command. The "definition" of an object of type FILEPKGCOMS is a list structure of the form ((COM . *COMPROPS*) (TYPE . *TYPEPROPS*)), where *COMPROPS* is a property list specifying how the name is defined as a file manager command by FILEPKGCOM (see the Defining New File Manager Commands section), and *TYPEPROPS* is a property list specifying how the name is defined as a file manager type by FILEPKGTYPE (see the Defining New File Manager Types section).

FILES [File Manager Type]

Used to access files. This file manager type is most useful for renaming files. The "definition" of a file is not a useful structure.

FILEVARS [File Manager Type]

Used to access Filevars (see the FileVars section).

FNS [File Manager Type]

Used to access function definitions.

I.S.OPRS [File Manager Type]

Used to access the definitions of iterative statement operators (see Chapter 9).

LISPXMACROS [File Manager Type]

Used to access programmer's assistant commands defined on the variables LISPXMACROS and LISPXHISTORYMACROS (see Chapter 13).

MACROS [File Manager Type]

Used to access macro definitions (see Chapter 10).

PROPS [File Manager Type]

Used to access objects stored on the property list of a symbol (see Chapter 2). When a property is changed or added, an object of type PROPS, with "name" (SYMBOL *PROPNAME*) is marked as being changed.

Note that some symbol properties are used to implement other file manager types. For example, the property MACRO implements the file manager type MACROS, the property ADVICE implements ADVICE, etc. This is indicated by putting the property PROPTYPE, with value of the file manager type on the property list of the property name. For example, (GETPROP 'MACRO 'PROPTYPE) => MACROS. When such a property is changed or added, an object of the corresponding file manager type is marked. If (GETPROP *PROPNAME* 'PROPTYPE) => IGNORE, the change is ignored. The FILE, FILEMAP, FILEDATES, etc. properties are all handled this way. (IGNORE cannot be the name of a file manager type implemented as a property).

RECORDS [File Manager Type]

Used to access record declarations (see Chapter 8).

RESOURCES [File Manager Type]

Used to access resources (see Chapter 12).

TEMPLATES [File Manager Type]

Used to access Masterscope templates (see Chapter 19).

USERMACROS [File Manager Type]

Used to access user edit macros (see Chapter 16).

VARS [File Manager Type]

Used to access top-level variable values.

Functions for Manipulating Typed Definitions

The functions described below can be used to manipulate typed definitions, without needing to know how the manipulations are done. For example, `(GETDEF 'FOO 'FNS)` will return the function definition of FOO, `(GETDEF 'FOO 'VARS)` will return the variable value of FOO, etc. All of the functions use the following conventions:

1. All functions which make destructive changes are undoable.

2. Any argument that expects a list of symbols will also accept a single symbol, operating as though it were enclosed in a list. For example, if the argument *FILES* should be a list of files, it may also be a single file.

3. *TYPE* is a file manager type. *TYPE* = NIL is equivalent to *TYPE* = FNS. The singular form of a file manager type is also recognized, e.g. *TYPE* = VAR is equivalent to *TYPE* = VARS.

4. *FILES* = NIL is equivalent to *FILES* = FILELST.

5. *SOURCE* is used to indicate the source of a definition, that is, where the definition should be found. *SOURCE* can be one of:

CURRENT Get the definition currently in effect.

SAVED Get the "saved" definition, as stored by SAVEDEF.

FILE Get the definition contained on the (first) file determined by WHEREIS.

WHEREIS is called with *FILES* = T, so that if the WHEREIS library package is loaded, the WHEREIS data base will be used to find the file containing the definition.

? Get the definition currently in effect if there is one, else the saved definition if there is one, otherwise the definition from a file determined by WHEREIS. Like specifying CURRENT, SAVED, and FILE in order, and taking the first definition that is found.

a file name
a list of file names Get the definition from the first of the indicated files that contains one.

NIL In most cases, giving *SOURCE* = NIL (or not specifying it at all) is the same as giving ?, to get either the current, saved, or filed definition. However, with HASDEF, *SOURCE* = NIL is interpreted as equal to *SOURCE* = CURRENT, which only tests if there is a current definition.

The operation of most of the functions described below can be changed or extended by modifying the appropriate properties for the corresponding file manager type using the function FILEPKGTYPE, described in the Defining New File Manager Types section.

(**GETDEF** *NAME TYPE SOURCE OPTIONS*) [Function]

Returns the definition of *NAME*, of type *TYPE*, from *SOURCE*. For most types, GETDEF returns the expression which would be pretty printed when dumping *NAME* as *TYPE*. For example, for *TYPE* = FNS, an EXPR definition is returned, for *TYPE* = VARS, the value of *NAME* is returned, etc.

OPTIONS is a list which specifies certain options:

NOERROR GETDEF causes an error if an appropriate definition cannot be found, unless *OPTIONS* is or contains NOERROR. In this case, GETDEF returns the value of the NULLDEF file manager type property (see the Defining New File Manager Types section), usually NIL.

a string If *OPTIONS* is or contains a string, that string will be returned if no definition is found (and NOERROR is not among the options). The caller can thus determine whether a definition was found, even for types for which NIL or NOBIND are acceptable definitions.

NOCOPY GETDEF returns a copy of the definition unless *OPTIONS* is or contains NOCOPY.

EDIT If *OPTIONS* is or contains EDIT, GETDEF returns a copy of the definition unless it is possible to edit the definition "in place." With some file manager types, such as functions, it is meaningful (and efficient) to edit the definition by destructively modifying the list structure, without calling PUTDEF. However, some file manager types (like records) need to be "installed" with PUTDEF after they are edited. The default EDITDEF (see the Defining New File Manager Types section) calls GETDEF with *OPTIONS* of (EDIT NOCOPY), so it doesn't use a copy unless it has to, and only calls PUTDEF if the result of editing is not EQUAL to the old definition.

NODWIM A FNS definition will be dwimified if it is likely to contain CLISP unless *OPTIONS* is or contains NODWIM.

(**PUTDEF** *NAME TYPE DEFINITION REASON*) [Function]

Defines *NAME* of type *TYPE* with *DEFINITION*. For *TYPE* = FNS, does a DEFINE; for *TYPE* = VARS, does a SAVESET, etc.

For *TYPE* = FILES, PUTDEF establishes the command list, notices *NAME*, and then calls MAKEFILE to actually dump the file *NAME*, copying functions if necessary from the "old" file (supplied as part of *DEFINITION*).

PUTDEF calls MARKASCHANGED (see the Mrking Changes section) to mark *NAME* as changed, giving a reason of *REASON*. If *REASON* is NIL, the default is DEFINED.

If *TYPE* = FNS, PUTDEF prints a warning if you try to redefine a function on the list UNSAFE.TO.MODIFY.FNS (see Chapter 10).

(**HASDEF** *NAME TYPE SOURCE SPELLFLG*) [Function]

Returns (OR *NAME* T) if *NAME* is the name of something of type *TYPE*. If not, attempts spelling correction if *SPELLFLG* = T, and returns the spelling-corrected *NAME*. Otherwise returns NIL. HASDEF for type FNS (or NIL) indicates that *NAME* has an editable source definition. If *NAME* is a function that exists on a file for which you have loaded only the compiled version and not the source, HASDEF returns NIL.

(HASDEF NIL *TYPE*) returns T if NIL has a valid definition.

If *SOURCE* = NIL, *HASDEF* interprets this as equal to *SOURCE* = CURRENT, which only tests if there is a current definition.

(**TYPESOF** *NAME POSSIBLETYPES IMPOSSIBLETYPES SOURCE*) [Function]

Returns a list of the types in *POSSIBLETYPES* but not in *IMPOSSIBLETYPES* for which *NAME* has a definition. FILEPKGTYPES is used if *POSSIBLETYPES* is NIL.

(**COPYDEF** *OLD NEW TYPE SOURCE OPTIONS*) [Function]

Defines *NEW* to have a copy of the definition of *OLD* by doing PUTDEF on a copy of the definition retrieved by (GETDEF *OLD TYPE SOURCE OPTIONS*). *NEW* is substituted for *OLD* in the copied definition, in a manner that may depend on the *TYPE*.

For example, (COPYDEF 'PDQ 'RST 'FILES) sets up RSTCOMS to be a copy of PDQCOMS, changes things like (VARS * PDQVARS) to be (VARS * RSTVARS) in RSTCOMS, and performs a MAKEFILE on RST such that the appropriate definitions get copied from PDQ.

COPYDEF disables the NOCOPY option of GETDEF, so *NEW* will always have a *copy* of the definition of *OLD*.

COPYDEF substitutes *NEW* for *OLD* throughout the definition of *OLD*. This is usually the right thing to do, but in some cases, e.g., where the old name appears within a quoted expression but was not used in the same context, you must re-edit the definition.

(**DELDEF** *NAME TYPE*) [Function]

Removes the definition of *NAME* as a *TYPE* that is currently in effect.

(**SHOWDEF** *NAME TYPE FILE*) [Function]

Prettyprints the definition of *NAME* as a *TYPE* to *FILE*. This shows you how *NAME* would be written to a file. Used by ADDTOFILES? (see the Storing Files section).

(**EDITDEF** *NAME TYPE SOURCE EDITCOMS*) [Function]

Edits the definition of *NAME* as a *TYPE*. Essentially performs

```
(PUTDEF NAME TYPE
   (EDITE (GETDEF NAME TYPE SOURCE)
         EDITCOMS))
```

(**SAVEDEF** *NAME TYPE DEFINITION*) [Function]

Sets the "saved" definition of *NAME* as a *TYPE* to *DEFINITION*. If *DEFINITION* = NIL, the current definition of *NAME* is saved.

If *TYPE* = FNS (or NIL), the function definition is saved on *NAME*'s property list under the property EXPR, or CODE (depending on the FNTYP of the function definition). If (GETD *NAME*) is non-NIL, but (FNTYP *FN*) = NIL, SAVEDEF saves the definition on the property name LIST. This can happen if a function was somehow defined with an illegal expr definition, such as (LAMMMMDA (X) ...).

If *TYPE* = VARS, the definition is stored as the value of the VALUE property of *NAME*. For other types, the definition is stored in an internal data structure, from where it can be retrieved by GETDEF or UNSAVEDEF.

(**UNSAVEDEF** *NAME TYPE*) [Function]

Restores the "saved" definition of *NAME* as a *TYPE*, making it be the current definition. Returns *PROP*.

If *TYPE* = FNS (or NIL), UNSAVEDEF unsaves the function definition from the EXPR property if any, else CODE, and returns the property name used. UNSAVEDEF also recognizes *TYPE* = EXPR, CODE, or LIST, meaning to unsave the definition only from the corresponding property only.

If DFNFLG is not T (see Chapter 10), the current definition of *NAME*, if any, is saved using SAVEDEF. Thus one can use UNSAVEDEF to switch back and forth between two definitions.

(**LOADDEF** *NAME TYPE SOURCE*) [Function]

Equivalent to (PUTDEF *NAME TYPE* (GETDEF *NAME TYPE SOURCE*)). LOADDEF is essentially a generalization of LOADFNS, e.g. it enables loading a single record declaration from a file. (LOADDEF *FN*) will give *FN* an EXPR definition, either obtained from its property list or a file, unless it already has one.

(**CHANGECALLERS** *OLD NEW TYPES FILES METHOD*) [Function]

Finds all of the places where *OLD* is used as any of the types in *TYPES* and changes those places to use *NEW*. For example, (CHANGECALLERS 'NLSETQ 'ERSETQ) will change all calls to NLSETQ to be calls to ERSETQ. Also changes occurrences of *OLD* to *NEW* inside the filecoms of any file, inside record declarations, properties, etc.

CHANGECALLERS attempts to determine if *OLD* might be used as more than one type; for example, if it is both a function and a record field. If so, rather than performing the transformation *OLD* -> *NEW* automatically, you are allowed to edit all of the places where *OLD* occurs. For each occurrence of *OLD*, you are asked whether you want to make the replacement. If you respond with anything except Yes or No, the editor is invoked on the expression containing that occurrence.

There are two different methods for determining which functions are to be examined. If *METHOD* = EDITCALLERS, EDITCALLERS is used to search *FILES* (see Chapter 16). If *METHOD* = MASTERSCOPE, then the Masterscope database is used instead. *METHOD* = NIL defaults to MASTERSCOPE if the value of the variable DEFAULTRENAMEMETHOD is MASTERSCOPE and a Masterscope database exists, otherwise it defaults to EDITCALLERS.

(**RENAME** *OLD NEW TYPES FILES METHOD*) [Function]

First performs (COPYDEF *OLD NEW TYPE*) for all *TYPE* inside *TYPES*. It then calls CHANGECALLERS to change all occurrences of *OLD* to *NEW*, and then "deletes" *OLD* with DELDEF. For example, if you have a function FOO which you now wish to call FIE, simply perform (RENAME 'FOO 'FIE), and FIE will be given FOO's definition, and all places that FOO are called will be changed to call FIE instead.

METHOD is interpreted the same as the *METHOD* argument to CHANGECALLERS, above.

(**COMPARE** $NAME_1$ $NAME_2$ *TYPE* $SOURCE_1$ $SOURCE_2$) [Function]

Compares the definition of $NAME_1$ with that of $NAME_2$, by calling COMPARELISTS (Chapter 3) on (GETDEF $NAME_1$ *TYPE* $SOURCE_1$) and (GETDEF $NAME_2$ *TYPE* $SOURCE_2$), which prints their differences on the terminal.

For example, if the current value of the variable A is (A B C (D E F) G), and the value of the variable B on the file <lisp>FOO is (A B C (D F E) G), then:

```
←(COMPARE 'A 'B 'VARS 'CURRENT '<lisp>FOO)
A from CURRENT and B from <lisp>TEST differ:
(E -> F) (F -> E)
T
```

(**COMPAREDEFS** *NAME TYPE SOURCES*) [Function]

Calls COMPARELISTS (Chapter 3) on all pairs of definitions of *NAME* as a *TYPE* obtained from the various *SOURCES* (interpreted as a list of source specifications).

Defining New File Manager Types

All manipulation of typed definitions in the file manager is done using the type-independent functions GETDEF, PUTDEF, etc. Therefore, to define a new file manager type, it is only necessary to specify (via the function FILEPKGTYPE) what these functions should do when dealing with a typed definition of the new type. Each file manager type has the following properties, whose values are functions or lists of functions:

These functions are defined to take a *TYPE* argument so that you may have the same function for more than one type.

GETDEF [File Manager Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *OPTIONS*, which should return the current definition of *NAME* as a type *TYPE*. Used by GETDEF (see the Functions for Manipulating Typed Definitions section), which passes its *OPTIONS* argument.

If there is no GETDEF property, a file manager command for dumping *NAME* is created (by MAKENEWCOM). This command is then used to write the definition of *NAME* as a type *TYPE* onto the file FILEPKG.SCRATCH (in Medley, this file is created on the {CORE} device). This expression is then read back in and returned as the current definition.

In some situations, the function HASDEF needs to call GETDEF to determine whether a definition exists. In this case, *OPTIONS* will include the symbol HASDEF, and it is permissable for a GETDEF function to return T or NIL, rather than creating a complex structure which will not be used.

NULLDEF [File Manager Type Property]

The value of the NULLDEF property is returned by GETDEF (see the Functions for Manipulating Typed Definitions section) when there is no definition and the NOERROR option is supplied. For example, the NULLDEF of VARS is NOBIND.

FILEGETDEF [File Manager Type Property]

This enables you to provide a way of obtaining definitions from a file that is more efficient than the default procedure used by GETDEF (see the Functions for Manipulating Typed Definitions section). Value is a function of four arguments, *NAME*, *TYPE*, *FILE*, and *OPTIONS*. The function is applied by GETDEF when it is determined that a typed definition is needed from a particular file. The function must open and search the given file and return any *TYPE* definition for *NAME* that it finds.

CANFILEDEF [File Manager Type Property]

If the value of this property is non-NIL, this indicates that definitions of this file manager type are not loaded when a file is loaded with LOADFROM (see the Loading Files section). The default is NIL. Initially, only FNS has this property set to non-NIL.

PUTDEF [File Manager Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *DEFINITION*, which should store *DEFINITION* as the definition of *NAME* as a type *TYPE*. Used by PUTDEF (see the Functions for Manipulating Typed Definitions section).

HASDEF [File Manager Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *SOURCE*, which should return (OR *NAME* T) if *NAME* is the name of something of type *TYPE*. *SOURCE* is as interpreted by HASDEF (see the Functions for Manipulating Typed Definitions section), which uses this property.

EDITDEF [File Manager Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *SOURCE*, and *EDITCOMS*, which should edit the definition of *NAME* as a type *TYPE* from the source *SOURCE*, interpreting the edit commands *EDITCOMS*. If sucessful, should return *NAME* (or a spelling-corrected *NAME*). If it returns NIL, the "default" editor is called. Used by EDITDEF (see the Functions for Manipulating Typed Definitions section).

DELDEF [File Manager Type Property]

Value is a function of two arguments, *NAME*, and *TYPE*, which removes the definition of *NAME* as a *TYPE* that is currently in effect. Used by DELDEF (see the Functions for Manipulating Typed Definitions section).

NEWCOM [File Manager Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *LISTNAME*, and *FILE*. Specifies how to make a new (instance of a) file manager command to dump *NAME*, an object of type *TYPE*. The function should return the new file manager command. Used by ADDTOFILE and SHOWDEF.

If *LISTNAME* is non-NIL, this means that you specified *LISTNAME* as the filevar in interaction with ADDTOFILES? (see the FileVars section).

If no NEWCOM is specified, the default is to call DEFAULTMAKENEWCOM, which will construct and return a command of the form (*TYPE NAME*). You can advise or redefine DEFAULTMAKENEWCOM .

WHENCHANGED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *REASON* when *NAME*, an instance of type *TYPE*, is changed or defined (see MARKASCHANGED, in the Marking Changes section). Used for various applications, e.g. when an object of type I.S.OPRS changes, it is necessary to clear the corresponding translatons from CLISPARRAY.

The WHENCHANGED functions are called before the object is marked as changed, so that it can, in fact, decide that the object is *not* to be marked as changed, and execute (RETFROM 'MARKASCHANGED).

The *REASON* argument passed to WHENCHANGED functions is either DEFINED or CHANGED.

WHENFILED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is added to *FILE*.

WHENUNFILED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is removed from *FILE*.

DESCRIPTION [File Manager Type Property]

Value is a string which describes instances of this type. For example, for type RECORDS, the value of DESCRIPTION is the string "record declarations".

The function FILEPKGTYPE is used to define new file manager types, or to change the properties of existing types. It is possible to redefine the attributes of system file manager types, such as FNS or PROPS.

(**FILEPKGTYPE** *TYPE* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N) [NoSpread Function]

Nospread function for defining new file manager types, or changing properties of existing file manager types. $PROP_i$ is one of the property names given above; VAL_i is the value to be given to that property. Returns *TYPE*.

(FILEPKGTYPE *TYPE PROP*) returns the value of the property *PROP*, without changing it.

(FILEPKGTYPE *TYPE*) returns a property list of all of the defined properties of *TYPE*, using the property names as keys.

Specifying *TYPE* as the symbol TYPE can be used to define one file manager type as a synonym of another. For example, (FILEPKGTYPE 'R 'TYPE 'RECORDS) defines R as a synonym for the file manager type RECORDS.

File Manager Commands

The basic mechanism for creating symbolic files is the function MAKEFILE (see the Storing Files section). For each file, the file manager has a data structure known as the "filecoms", which specifies what typed descriptions are contained in the file. A filecoms is a list of file manager commands, each of which specifies objects of a certain file manager type which should be dumped. For example, the filecoms

```
((FNS FOO)
   (VARS FOO BAR BAZ)
   (RECORDS XYZZY))
```

has a FNS, a VARS, and a RECORDS file manager command. This filecoms specifies that the function definition for FOO, the variable values of FOO, BAR, and BAZ, and the record declaration for XYZZY should be dumped.

By convention, the filecoms of a file *X* is stored as the value of the symbol *X*COMS. For example, (MAKEFILE 'FOO.;27) will use the value of FOOCOMS as the filecoms. This variable can be directly manipulated, but the file manager contains facilities which make constructing and updating filecoms easier, and in some cases automatic (see the Functions for Manipulating File Command Lists section).

A file manager command is an instruction to MAKEFILE to perform an explicit, well-defined operation, usually printing an expression. Usually there is a one-to-one correspondence between file manager types and file manager commands; for each file manager type, there is a file manager command which is used for writing objects of that type to a file, and each file manager command is used to write objects of a particular type. However, in some cases, the same file manager type can be dumped by several different file manager commands. For example, the file manager commands PROP, IFPROP, and PROPS all dump out objects with the file manager type PROPS. This means if you change an object of file manager type PROPS via EDITP, a typed-in call to PUTPROP, or via an explicit call to MARKASCHANGED, this object can be written out with any of the above three commands. Thus, when the file manager attempts to determine whether this typed object is contained on a particular file, it must look at instances of all three file manager commands PROP, IFPROP, and PROPS, to see if the corresponding atom and property are specified. It is also permissible for a single file manager command to dump several different file manager types. For example, you can define a file manager command which dumps both a function definition and its macro. Conversely, some file manager comands do not dump any file manager types at all, such as the E command.

For each file manager command, the file manager must be able to determine what typed definitions the command will cause to be printed so that the file manager can determine on what file (if any) an object of a given type is contained (by searching through the filecoms). Similarly, for each file manager type, the file manager must be able to construct a command that will print out an object of that type. In other words, the file manager must be able to map file manager commands into file manager types, and vice versa. Information can be provided to the file manager about a particular file manager command via the function FILEPKGCOM (see the Defining New File Manager Commands section), and information about a particular file manager type via the function FILEPKGTYPE (see the prior section). In the absence of other information, the default is simply that a file manager command of the form (*X NAME*) prints out the definition of *NAME* as a type *X*, and, conversely, if *NAME* is an object of type *X*, then *NAME* can be written out by a command of the form (*X NAME*).

If a file manager function is given a command or type that is not defined, it attempts spelling correction using FILEPKGCOMSPLST as a spelling list (unless DWIMFLG or NOSPELLFLG = NIL; see Chapter 20). If successful, the corrected version of the list of file manager commands is written (again) on the output file, since at this point, the uncorrected list of file manager commands would already have been printed on the output file. When the file is loaded, this will result in *FILE*COMS being reset, and may cause a message to be printed, e.g., (FOOCOMS RESET). The value of FOOCOMS would then be the corrected version. If the spelling correction is unsuccessful, the file manager functions generate an error, BAD FILE PACKAGE COMMAND.

File package commands can be used to save on the output file definitions of functions, values of variables, property lists of atoms, advised functions, edit macros, record declarations, etc. The interpretation of each file manager command is documented in the following sections.

(**USERMACROS** *SYMBOL*$_1$ ... *SYMBOL*$_N$) [File Manager Command]

Each symbol *SYMBOL*$_i$ is the name of a user edit macro. Writes expressions to add the edit macro definitions of *SYMBOL*$_i$ to USERMACROS, and adds the names of the commands to the appropriate spelling lists.

If *SYMBOL*$_i$ is not a user macro, a warning message "no EDIT MACRO for *SYMBOL*$_i$" is printed.

Functions and Macros

(**FNS** *FN*$_1$ ... *FN*$_N$) [File Manager Command]

Writes a DEFINEQ expression with the function definitions of *FN*$_1$ ... *FN*$_N$.

You should never print a DEFINEQ expression directly onto a file (by using the P file manager command, for example), because MAKEFILE generates the filemap of function definitions from the FNS file manager commands (see the File Maps section).

(**ADVISE** *FN*$_1$ ... *FN*$_N$) [File Manager Command]

For each function *FN*$_i$, writes expressions to reinstate the function to its advised state when the file is loaded. See Chapter 15.

When advice is applied to a function programmatically or by hand, it is additive. That is, if a function already has some advice, further advice is added to the already-existing advice. However, when advice is applied to a function as a result of loading a file with an ADVISE file manager command, the new advice replaces any earlier advice. ADVISE works this way to prevent problems with loading different versions of the same advice. If you really want to apply additive advice, a file manager command such as (P (ADVISE ...)) should be used (see the Miscellaneous File Manager Commands section).

(**ADVICE** *FN*$_1$ ... *FN*$_N$) [File Manager Command]

For each function *FN*$_i$, writes a PUTPROPS expression which will put the advice back on the property list of the function. You can then use READVISE (see Chapter 15) to reactivate the advice.

(**MACROS** *SYMBOL*$_1$ ... *SYMBOL*$_N$) [File Manager Command]

Each *SYMBOL*$_i$ is a symbol with a MACRO definition (and/or a DMACRO, 10MACRO, etc.). Writes out an expression to restore all of the macro properties for each *SYMBOL*$_i$, embedded in a DECLARE: EVAL@COMPILE so the macros will be defined when the file is compiled. See Chapter 10.

Variables

(**VARS** VAR_1 ... VAR_N) [File Manager Command]

For each VAR_i, writes an expression to set its top level value when the file is loaded. If VAR_i is atomic, VARS writes out an expression to set VAR_i to the top-level value it had at the time the file was written. If VAR_i is non-atomic, it is interpreted as (*VAR FORM*), and VARS write out an expression to set *VAR* to the value of *FORM* (evaluated when the file is loaded).

VARS prints out expressions using RPAQQ and RPAQ, which are like SETQQ and SETQ except that they also perform some special operations with respect to the file manager (see the Functions Used within Source Files section).

VARS cannot be used for putting arbitrary variable values on files. For example, if the value of a variable is an array (or many other data types), a symbol which represents the array is dumped in the file instead of the array itself. The HORRIBLEVARS file manager command provides a way of saving and reloading variables whose values contain re-entrant or circular list structure, user data types, arrays, or hash arrays.

(**INITVARS** VAR_1 ... VAR_N) [File Manager Command]

INITVARS is used for initializing variables, setting their values only when they are currently NOBIND. A variable value defined in an INITVARS command will not change an already established value. This means that re-loading files to get some other information will not automatically revert to the initialization values.

The format of an INITVARS command is just like VARS. The only difference is that if VAR_i is atomic, the current value is not dumped; instead NIL is defined as the initialization value. Therefore, (INITVARS FOO (FUM 2)) is the same as (VARS (FOO NIL)(FUM 2)), if FOO and FUM are both NOBIND.

INITVARS writes out an RPAQ? expression on the file instead of RPAQ or RPAQQ.

(**ADDVARS** (VAR_1 . LST_1)...(VAR_N . LST_N)) [File Manager Command]

For each (VAR_i . LST_i), writes an ADDTOVAR (see the Functions Used Within Source Files section) to add each element of LST_i to the list that is the value of VAR_i at the time the file is loaded. The new value of VAR_i will be the union of its old value and LST_i. If the value of VAR_i is NOBIND, it is first set to NIL.

For example, (ADDVARS (DIRECTORIES LISP LISPUSERS)) will add LISP and LISPUSERS to the value of DIRECTORIES.

If LST_i is not specified, VAR_i is initialized to NIL if its current value is NOBIND. In other words, (ADDVARS (*VAR*)) will initialize *VAR* to NIL if *VAR* has not previously been set.

(**APPENDVARS** (VAR_1 . LST_1) ... (VAR_N . LST_N)) [File Manager Command]

The same as ADDVARS, except that the values are added to the end of the lists (using APPENDTOVAR, in the Functions Used Within Source Files section), rather than at the beginning.

(**UGLYVARS** VAR_1 ... VAR_N) [File Manager Command]

Like VARS, except that the value of each VAR_i may contain structures for which READ is not an inverse of PRINT, e.g. arrays, readtables, user data types, etc. Uses HPRINT (see Chapter 25).

(**HORRIBLEVARS** VAR_1 ... VAR_N) [File Manager Command]

Like UGLYVARS, except structures may also contain circular pointers. Uses HPRINT (see Chapter 25). The values of VAR_1 ... VAR_N are printed in the same operation, so that they may contain pointers to common substructures.

UGLYVARS does not do any checking for circularities, which results in a large speed and internal-storage advantage over HORRIBLEVARS. Thus, if it is known that the data structures do *not* contain circular pointers, UGLYVARS should be used instead of HORRIBLEVARS.

(**ALISTS** (VAR_1 KEY_1 KEY_2 ...) ... (VAR_N KEY_3 KEY_4 ...)) [File Manager Command]

VAR_i is a variable whose value is an association list, such as EDITMACROS, BAKTRACELST, etc. For each VAR_i, ALISTS writes out expressions which will restore the values associated with the specified keys. For example, (ALISTS (BREAKMACROS BT BTV)) will dump the definition for the BT and BTV commands on BREAKMACROS.

Some association lists (USERMACROS, LISPXMACROS, etc.) are used to implement other file manager types, and they have their own file manager commands.

(**SPECVARS** VAR_1 ... VAR_N) [File Manager Command]

(**LOCALVARS** VAR_1 ... VAR_N) [File Manager Command]

(**GLOBALVARS** VAR_1 ... VAR_N) [File Manager Command]

Outputs the corresponding compiler declaration embedded in a DECLARE: DOEVAL@COMPILE DONTCOPY. See Chapter 18.

(**CONSTANTS** VAR_1 ... VAR_N) [File Manager Command]

Like VARS, for each VAR_i writes an expression to set its top level value when the file is loaded. Also writes a CONSTANTS expression to declare these variables as constants (see Chapter 18). Both of these expressions are wrapped in a (DECLARE: EVAL@COMPILE ...) expression, so they can be used by the compiler.

Like VARS, VAR_i can be non-atomic, in which case it is interpreted as (*VAR FORM*), and passed to CONSTANTS (along with the variable being initialized to *FORM*).

Symbol Properties

(**PROP** *PROPNAME* $SYMBOL_1$ ... $SYMBOL_N$) [File Manager Command]

Writes a PUTPROPS expression to restore the value of the *PROPNAME* property of each symbol $SYMBOL_i$ when the file is loaded.

If *PROPNAME* is a list, expressions will be written for each property on that list. If *PROPNAME* is the symbol ALL, the values of all user properties (on the property list of each $SYMBOL_i$) are saved. SYSPROPS is a list of properties used by system functions. Only properties *not* on that list are dumped when the ALL option is used.

If $SYMBOL_i$ does not have the property *PROPNAME* (as opposed to having the property with value NIL), a warning message "NO *PROPNAME* PROPERTY FOR $SYMBOL_i$" is printed. The command IFPROP can be used if it is not known whether or not an atom will have the corresponding property.

(**IFPROP** *PROPNAME* $SYMBOL_1$ ... $SYMBOL_N$) [File Manager Command]

Same as the PROP file manager command, except that it only saves the properties that actually appear on the property list of the corresponding atom. For example, if FOO1 has property PROP1 and PROP2, FOO2 has PROP3, and FOO3 has property PROP1 and PROP3, then (IFPROP (PROP1 PROP2 PROP3) FOO1 FOO2 FOO3) will save only those five property values.

(**PROPS** ($SYMBOL_1$ $PROPNAME_1$)...($SYMBOL_N$ $PROPNAME_N$)) [File Manager Command]

Similar to PROP command. Writes a PUTPROPS expression to restore the value of $PROPNAME_i$ for each $SYMBOL_i$ when the file is loaded.

As with the PROP command, if $SYMBOL_i$ does not have the property *PROPNAME* (as opposed to having the property with NIL value), a warning message "NO $PROPNAME_i$ PROPERTY FOR $SYMBOL_i$" is printed.

Miscellaneous File Manager Commands

(**RECORDS** REC_1 ... REC_N) [File Manager Command]

Each REC_i is the name of a record (see Chapter 8). Writes expressions which will redeclare the records when the file is loaded.

(**INITRECORDS** REC_1 ... REC_N) [File Manager Command]

Similar to RECORDS, INITRECORDS writes expressions on a file that will, when loaded, perform whatever initialization/allocation is necessary for the indicated records.

However, the record declarations themselves are not written out. This facility is useful for building systems on top of Interlisp, in which the implementor may want to eliminate the record declarations from a production version of the system, but the allocation for these records must still be done.

(**LISPXMACROS** *SYMBOL*$_1$ ... *SYMBOL*$_N$) [File Manager Command]

Each *SYMBOL*$_i$ is defined on LISPXMACROS or LISPXHISTORYMACROS (see Chapter 13). Writes expressions which will save and restore the definition for each macro, as well as making the necessary additions to LISPXCOMS

(**I.S.OPRS** *OPR*$_1$ ... *OPR*$_N$) [File Manager Command]

Each *OPR*$_i$ is the name of a user-defined i.s.opr (see Chapter 9). Writes expressions which will redefine the i.s.oprs when the file is loaded.

(**RESOURCES** *RESOURCE*$_1$ ... *RESOURCE*$_N$) [File Manager Command]

Each *RESOURCES*$_i$ is the name of a resource (see Chapter 12). Writes expressions which will redeclare the resource when the file is loaded.

(**INITRESOURCES** *RESOURCE*$_1$ ... *RESOURCE*$_N$) [File Manager Command]

Parallel to INITRECORDS, INITRESOURCES writes expressions on a file to perform whatever initialization / allocation is necessary for the indicated resources, without writing the resource declaration itself.

(**COURIERPROGRAMS** *NAME*$_1$ ... *NAME*$_N$) [File Manager Command]

Each *NAME*$_i$ is the name of a Courier program (see Chapter 31). Writes expressions which will redeclare the Courier program when the file is loaded.

(**TEMPLATES** *SYMBOL*$_1$ ... *SYMBOL*$_N$) [File Manager Command]

Each *SYMBOL*$_i$ is a symbol which has a Masterscope template (see Chapter 19). Writes expressions which will restore the templates when the file is loaded.

(**FILES** *FILE*$_1$ ... *FILE*$_N$) [File Manager Command]

Used to specify auxiliary files to be loaded in when the file is loaded. Dumps an expression calling FILESLOAD (see the Loading Files section), with *FILE*$_1$ ... *FILE*$_N$ as the arguments. FILESLOAD interprets *FILE*$_1$ ... *FILE*$_N$ as files to load, possibly interspersed with lists used to specify certain loading options.

(**FILEPKGCOMS** *SYMBOL*$_1$ ... *SYMBOL*$_N$) [File Manager Command]

Each symbol *SYMBOL*$_i$ is either the name of a user-defined file manager command or a user-defined file manager type (or both). Writes expressions which will restore each command / type.

If $SYMBOL_i$ is not a file manager command or type, a warning message "no FILE PACKAGE COMMAND for $SYMBOL_i$" is printed.

(*** .** *TEXT*) [File Manager Command]

Used for inserting comments in a file. The file manager command is simply written on the output file; it will be ignored when the file is loaded.

If the first element of *TEXT* is another *, a form-feed is printed on the file before the comment.

(**P** EXP_1 ... EXP_N) [File Manager Command]

Writes each of the expressions EXP_1 ... EXP_N on the output file, where they will be evaluated when the file is loaded.

(**E** $FORM_1$ ... $FORM_N$) [File Manager Command]

Each of the forms $FORM_1$ ... $FORM_N$ is evaluated at *output* time, when MAKEFILE interpretes this file manager command.

(**COMS** COM_1 ... COM_N) [File Manager Command]

Each of the commands COM_1 ... COM_N is interpreted as a file manager command.

(**ORIGINAL** COM_1 ... COM_N) [File Manager Command]

Each of the commands COM_i will be interpreted as a file manager command without regard to any file manager macros (as defined by the MACRO property of the FILEPKGCOM function, in the Defining New File Manager Commands section). Useful for redefining a built-in file manager command in terms of itself.

Some of the "built-in" file manager commands are defined by file manager macros, so interpreting them (or new user-defined file manager commands) with ORIGINAL will fail. ORIGINAL was never intended to be used outside of a file manager command macro.

DECLARE:

(**DECLARE:** . *FILEPKGCOMS/FLAGS*) [File Manager Command]

Normally expressions written onto a symbolic file are evaluated when loaded; copied to the compiled file when the symbolic file is compiled (see Chapter 18); and not evaluated at compile time. DECLARE: allows you to override these defaults.

FILEPKGCOMS/FLAGS is a list of file manager commands, possibly interspersed with "tags". The output of those file manager commands within *FILEPKGCOMS/FLAGS* is embedded in a DECLARE: expression, along with any tags that are specified. For example, (DECLARE: EVAL@COMPILE DONTCOPY (FNS ...) (PROP ...)) would produce (DECLARE: EVAL@COMPILE DONTCOPY (DEFINEQ ...) (PUTPROPS

...)). DECLARE: is *defined* as an nlambda nospread function, which processes its arguments by evaluating or not evaluating each expression depending on the setting of internal state variables. The initial setting is to evaluate, but this can be overridden by specifying the DONTEVAL@LOAD tag.

DECLARE: expressions are specially processed by the compiler. For the purposes of compilation, DECLARE: has two principal applications: to specify forms that are to be evaluated at compile time, presumably to affect the compilation, e.g., to set up macros; and/or to indicate which expressions appearing in the symbolic file are *not* to be copied to the output file. (Normally, expressions are *not* evaluated and *are* copied.) Each expression in CDR of a DECLARE: form is either evaluated/not-evaluated and copied/not-copied depending on the settings of two internal state variables, initially set for copy and not-evaluate. These state variables can be reset for the remainder of the expressions in the DECLARE: by means of the tags DONTCOPY, EVAL@COMPILE, etc.

The tags are:

EVAL@LOAD

DOEVAL@LOAD Evaluate the following forms when the file is loaded (unless overridden by DONTEVAL@LOAD).

DONTEVAL@LOAD Do not evaluate the following forms when the file is loaded.

EVAL@LOADWHEN This tag can be used to provide conditional evaluation. The value of the expression immediately following the tag determines whether or not to evaluate subsequent expressions when loading. ... EVAL@LOADWHEN T ... is equivalent to ... EVAL@LOAD ...

COPY

DOCOPY When compiling, copy the following forms into the compiled file.

DONTCOPY When compiling, do not copy the following forms into the compiled file.

Note: If the file manager commands following DONTCOPY include record declarations for datatypes, or records with initialization forms, it is necessary to include a INITRECORDS file manager command (see the prior section) outside of the DONTCOPY form so that the initialization information is copied. For example, if FOO was defined as a datatype,

```
(DECLARE: DONTCOPY (RECORDS FOO))
(INITRECORDS FOO)
```

would copy the data type declaration for FOO, but would not copy the whole record declaration.

COPYWHEN When compiling, if the next form evaluates to non-NIL, copy the following forms into the compiled file.

EVAL@COMPILE
DOEVAL@COMPILE When compiling, evaluate the following forms.

DONTEVAL@COMPILE When compiling, do not evaluate the following forms.

EVAL@COMPILEWHEN When compiling, if the next form evaluates to non-NIL, evaluate the following forms.

FIRST For expressions that are to be copied to the compiled file, the tag FIRST can be used to specify that the following expressions in the DECLARE: are to appear at the front of the compiled file, before anything else except the FILECREATED expressions (see the Symbolic File Format section). For example, (DECLARE: COPY FIRST (P (PRINT *MESS1* T)) NOTFIRST (P (PRINT *MESS2* T))) will cause (PRINT *MESS1* T) to appear first in the compiled file, followed by any functions, then (PRINT MESS2 T).

NOTFIRST Reverses the effect of FIRST.

The value of DECLARETAGSLST is a list of all the tags used in DECLARE: expressions. If a tag not on this list appears in a DECLARE: file manager command, spelling correction is performed using DECLARETAGSLST as a spelling list.

Note that the function LOADCOMP (see the Loading Files section) provides a convenient way of obtaining information from the DECLARE: expressions in a file, without reading in the entire file. This information may be used for compiling other files.

(**BLOCKS** *BLOCK*$_1$ ... *BLOCK*$_N$) [File Manager Command]

For each *BLOCK*$_i$, writes a DECLARE: expression which the block compile functions interpret as a block declaration. See Chapter 18.

Exporting Definitions

When building a large system in Interlisp, it is often the case that there are record definitions, macros and the like that are needed by several different system files when running, analyzing and compiling the source code of the system, but which are not needed for running the compiled code. By using the DECLARE: file manager command with tag DONTCOPY (see the prior section), these definitions can be kept out of the compiled files, and hence out of the system constructed by loading the compiled files files into Interlisp. This saves loading time, space in the resulting system, and whatever other overhead might be incurred by keeping those definitions around, e.g., burden on the record package to consider more possibilities in translating record accesses, or conflicts between system record fields and user record fields.

However, if the implementor wants to debug or compile code in the resulting system, the definitions are needed. And even if the definitions *had* been copied to the compiled files, a similar problem arises if one wants to work on system code in a regular Interlisp environment where none of the system files had been loaded. One could mandate that any definition needed by more than one file in the system should reside on a distinguished file of definitions, to be loaded into any environment where the system files are worked on. Unfortunately, this would keep the definitions away from where they logically belong. The EXPORT mechanism is designed to solve this problem.

To use the mechanism, the implementor identifies any definitions needed by files other than the one in which the definitions reside, and wraps the corresponding file manager commands in the EXPORT file manager command. Thereafter, GATHEREXPORTS can be used to make a single file containing all the exports.

(**EXPORT** *COM*$_1$ . . . *COM*$_N$) [File Manager Command]

This command is used for "exporting" definitions. Like COM, each of the commands *COM*$_1$ . . . *COM*$_N$ is interpreted as a file manager command. The commands are also flagged in the file as being "exported" commands, for use with GATHEREXPORTS.

(**GATHEREXPORTS** *FROMFILES TOFILE FLG*) [Function]

FROMFILES is a list of files containing EXPORT commands. GATHEREXPORTS extracts all the exported commands from those files and produces a loadable file *TOFILE* containing them. If *FLG* = EVAL, the expressions are evaluated as they are gathered; i.e., the exports are effectively loaded into the current environment as well as being written to *TOFILE*.

(**IMPORTFILE** *FILE RETURNFLG*) [Function]

If *RETURNFLG* is NIL, this loads any exported definitions from *FILE* into the current environment. If *RETURNFLG* is T, this returns a list of the exported definitions (evaluable expressions) without actually evaluating them.

(**CHECKIMPORTS** *FILES NOASKFLG*) [Function]

Checks each of the files in *FILES* to see if any exists in a version newer than the one from which the exports in memory were taken (GATHEREXPORTS and IMPORTFILE note the creation dates of the files involved), or if any file in the list has not had its exports loaded at all. If there are any such files, you are asked for permission to IMPORTFILE each such file. If *NOASKFLG* is non-NIL, IMPORTFILE is performed without asking.

For example, suppose file FOO contains records R1, R2, and R3, macros BAR and BAZ, and constants CON1 and CON2. If the definitions of R1, R2, BAR, and BAZ are needed by files other than FOO, then the file commands for FOO might contain the command

```
(DECLARE: EVAL@COMPILE DONTCOPY
   (EXPORT (RECORDS R1 R2)
           (MACROS BAR BAZ))
   (RECORDS R3)
   (CONSTANTS BAZ))
```

None of the commands inside this DECLARE: would appear on FOO's compiled file, but (GATHEREXPORTS '(FOO) 'MYEXPORTS) would copy the record definitions for R1 and R2 and the macro definitions for BAR and BAZ to the file MYEXPORTS.

FileVars

In each of the file manager commands described above, if the symbol * follows the command type, the form following the *, i.e., CADDR of the command, is evaluated and its value used in executing the command, e.g., (FNS * (APPEND FNS1 FNS2)). When this form is a symbol, e.g. (FNS * FOOFNS), we say that the variable is a "filevar". Note that (COMS * *FORM*) provides a way of *computing* what should be done by MAKEFILE.

Example:

```
← (SETQ FOOFNS '(FOO1 FOO2 FOO3))
       (FOO1 FOO2 FOO3)

← (SETQ FOOCOMS
       '((FNS * FOOFNS)
       (VARS FIE)
       (PROP MACRO FOO1 FOO2)
       (P (MOVD 'FOO1 'FIE1))]

← (MAKEFILE 'FOO)
```

would create a file FOO containing:

```
(FILECREATED "time and date the file was made" . "other information")
       (PRETTYCOMPRINT FOOCOMS)
       (RPAQQ FOOCOMS ((FNS * FOOFNS) ...)
       (RPAQQ FOOFNS (FOO1 FOO3 FOO3))
       (DEFINEQ "definitions of FOO1, FOO2, and FOO3")
       (RPAQQ FIE "value of FIE")
       (PUTPROPS FOO1 MACRO PROPVALUE)
       (PUTPROPS FOO2 MACRO PROPVALUE)
       (MOVD (QUOTE FOO1) (QUOTE FIE1))
       STOP
```

For the PROP and IFPROP commands (see the Litatom Properties section), the * follows the property name instead of the command, e.g., (PROP MACRO * FOOMACROS). Also, in the form (* * comment ...), the word comment is not treated as a filevar.

Defining New File Manager Commands

A file manager command is defined by specifying the values of certain properties. You can specify the various attributes of a file manager command for a new command, or respecify them for an existing command. The following properties are used:

MACRO [File Manager Command Property]

Defines how to dump the file manager command. Used by MAKEFILE. Value is a pair (*ARGS* . *COMS*). The "arguments" to the file manager command are substituted for *ARGS* throughout *COMS*, and the result treated as a list of file manager commands. For example, following (FILEPKGCOM 'FOO 'MACRO '((X Y) . *COMS*)), the file manager command (FOO A B) will cause A to be substituted for X and B for Y throughout *COMS*, and then *COMS* treated as a list of commands.

The substitution is carried out by SUBPAIR (see Chapter 3), so that the "argument list" for the macro can also be atomic. For example, if (X . *COMS*) was used instead of ((X Y) . *COMS*), then the command (FOO A B) would cause (A B) to be substituted for X throughout *COMS*.

Filevars are evaluated *before* substitution. For example, if the symbol * follows *NAME* in the command, CADDR of the command is evaluated substituting in *COMS*.

ADD [File Manager Command Property]

Specifies how (if possible) to add an instance of an object of a particular type to a given file manager command. Used by ADDTOFILE. Value is *FN*, a function of three arguments, *COM*, a file manager command CAR of which is EQ to *COMMANDNAME*, *NAME*, a typed object, and *TYPE*, its type. *FN* should return T if it (undoably) adds *NAME* to *COM*, NIL if not. If no ADD property is specified, then the default is (1) if (CAR *COM*) = *TYPE* and (CADR *COM*) = *, and (CADDR *COM*) is a filevar (i.e. a literal atom), add *NAME* to the value of the filevar, or (2) if (CAR *COM*) = *TYPE* and **(**CADR *COM*) is not *, add *NAME* to (CDR *COM*).

Actually, the function is given a fourth argument, *NEAR*, which if non-NIL, means the function should try to add the item after *NEAR*. See discussion of ADDTOFILES?, in the Storing Files section.

DELETE [File Manager Command Property]

Specifies how (if possible) to delete an instance of an object of a particular type from a given file manager command. Used by DELFROMFILES. Value is *FN*, a function of three arguments, *COM*, *NAME*, and *TYPE*, same as for ADD. *FN* should return T if it (undoably) deletes *NAME* from *COM*, NIL if not. If no DELETE property is specified, then the default is either (CAR *COM*) = *TYPE* and (CADR *COM*) = *, and (CADDR *COM*) is a filevar (i.e. a literal atom), and *NAME* is contained in the value of the filevar, then remove *NAME* from the filevar, or if (CAR *COM*) = *TYPE* and (CADR *COM*) is not *, and *NAME* is contained in (CDR *COM*), then remove *NAME* from (CDR *COM*).

If *FN* returns the value of ALL, it means that the command is now "empty", and can be deleted entirely from the command list.

CONTENTS [File Manager Command Property]

CONTAIN [File Manager Command Property]

Determines whether an instance of an object of a given type is contained in a given file manager command. Used by WHEREIS and INFILECOMS?. Value is *FN*, a function of three arguments, *COM*, a file manager command CAR of which is EQ to *COMMANDNAME*, *NAME*, and *TYPE*. The interpretation of *NAME* is as follows: if *NAME* is NIL, *FN* should return a list of elements of type *TYPE* contained in *COM*. If *NAME* is T, *FN* should return T if there are *any* elements of type *TYPE* in *COM*. If *NAME* is an atom other than T or NIL, return T if *NAME* of type *TYPE* is contained in *COM*. Finally, if *NAME* is a list, return a list of those elements of type *TYPE* contained in *COM* that are also contained in *NAME*.

It is sufficient for the CONTENTS function to simply return the list of items of type *TYPE* in command *COM*, i.e. it can in fact ignore the *NAME* argument. The *NAME* argument is supplied mainly for those situations where producing the entire list of items involves significantly more computation or creates more storage than simply determining whether a particular item (or any item) of type *TYPE* is contained in the command.

If a CONTENTS property is specified and the corresponding function application returns NIL and (CAR *COM*) = *TYPE*, then the operation indicated by *NAME* is performed on the value of (CADDR *COM*), if (CADR *COM*) = *, otherwise on (CDR *COM*). In other words, by specifying a CONTENTS property that returns NIL, e.g. the function NILL, you specify that a file manager command of name FOO produces objects of file manager type FOO and only objects of type FOO.

If the CONTENTS property is not provided, the command is simply expanded according to its MACRO definition, and each command on the resulting command list is then interrogated.

If *COMMANDNAME* is a file manager command that is used frequently, its expansion by the various parts of the system that need to interrogate files can result in a large number of CONSes and garbage collections. By informing the file manager as to what this command actually does and does not produce via the CONTENTS property, this expansion is avoided. For example, suppose you have a file manager command called GRAMMARS which dumps various property lists but no functions. The file manager could ignore this command when seeking information about FNS.

The function FILEPKGCOM is used to define new file manager commands, or to change the properties of existing commands. It is possible to redefine the attributes of system file manager commands, such as FNS or PROPS, and to cause unpredictable results.

(**FILEPKGCOM** *COMMANDNAME* $PROP_1$ VAL_1 . . . $PROP_N$ VAL_N) [NoSpread Function]

Nospread function for defining new file manager commands, or changing properties of existing file manager commands. $PROP_i$ is one of of the property names described above; VAL_i is the value to be given that property of the file manager command *COMMANDNAME*. Returns *COMMANDNAME*.

(FILEPKGCOM *COMMANDNAME PROP*) returns the value of the property *PROP*, without changing it.

(FILEPKGCOM *COMMANDNAME*) returns a property list of all of the defined properties of *COMMANDNAME*, using the property names as keys.

Specifying *TYPE* as the symbol COM can be used to define one file manager command as a synonym of another. For example, (FILEPKGCOM 'INITVARIABLES 'COM 'INITVARS) defines INITVARIABLES as a synonym for the file manager command INITVARS.

Functions for Manipulating File Command Lists

The following functions may be used to manipulate filecoms. The argument *COMS* does *not* have to correspond to the filecoms for some file. For example, *COMS* can be the list of commands generated as a result of expanding a user-defined file manager command.

The following functions will accept a file manager command as a valid value for their *TYPE* argument, even if it does not have a corresponding file manager type. User-defined file manager commands are expanded as necessary.

(**INFILECOMS?** *NAME TYPE COMS*) [Function]

COMS is a list of file manager commands, or a variable whose value is a list of file manager commands. *TYPE* is a file manager type. INFILECOMS? returns T if *NAME* of type *TYPE* is "contained" in *COMS*.

If *NAME* = NIL, INFILECOMS? returns a list of all elements of type *TYPE*.

If *NAME* = T, INFILECOMS? returns T if there are *any* elements of type *TYPE* in *COMS*.

(**ADDTOFILE** *NAME TYPE FILE NEAR LISTNAME*) [Function]

Adds *NAME* of type *TYPE* to the file manager commands for *FILE*. If *NEAR* is given and it is the name of an item of type *TYPE* already on *FILE*, then *NAME* is added to the command that dumps *NEAR*. If *LISTNAME* is given and is the name of a list of items of *TYPE* items on *FILE*, then *NAME* is added to that list. Uses ADDTOCOMS and MAKENEWCOM. Returns *FILE*. ADDTOFILE is undoable.

(**DELFROMFILES** *NAME TYPE FILES*) [Function]

Deletes all instances of *NAME* of type *TYPE* from the filecoms for each of the files on *FILES*. If *FILES* is a non-NIL symbol, (LIST *FILES*) is used. *FILES* = NIL defaults to FILELST. Returns a list of files from which *NAME* was actually removed. Uses DELFROMCOMS. DELFROMFILES is undoable.

Deleting a function will also remove the function from any BLOCKS declarations in the filecoms.

(**ADDTOCOMS** *COMS NAME TYPE NEAR LISTNAME*) [Function]

Adds *NAME* as a *TYPE* to *COMS*, a list of file manager commands or a variable whose value is a list of file manager commands. Returns NIL if ADDTOCOMS was unable to find a command appropriate for adding *NAME* to *COMS*. *NEAR* and *LISTNAME* are described in the discussion of ADDTOFILE. ADDTOCOMS is undoable.

The exact algorithm for adding commands depends the particular command itself. See discussion of the ADD property, in the description of FILEPKGCOM.

ADDTOCOMS will not attempt to add an item to any command which is inside of a DECLARE: unless you specified a specific name via the LISTNAME or NEAR option of ADDTOFILES?.

(**DELFROMCOMS** *COMS NAME TYPE*) [Function]

Deletes *NAME* as a *TYPE* from *COMS*. Returns NIL if DELFROMCOMS was unable to modify *COMS* to delete *NAME*. DELFROMCOMS is undoable.

(**MAKENEWCOM** *NAME TYPE*) [Function]

Returns a file manager command for dumping *NAME* of type *TYPE*. Uses the procedure described in the discussion of NEWCOM, in the Defining New File Manager Types section.

(**MOVETOFILE** *TOFILE NAME TYPE FROMFILE*) [Function]

Moves the definition of *NAME* as a *TYPE* from *FROMFILE* to *TOFILE* by modifying the file commands in the appropriate way (with DELFROMFILES and ADDTOFILE).

Note that if *FROMFILE* is specified, the definition will be retrieved from that file, even if there is another definition currently in your environment.

(**FILECOMSLST** *FILE TYPE*) [Function]

Returns a list of all objects of type *TYPE* in *FILE*.

(**FILEFNSLST** *FILE*) [Function]

Same as (FILECOMSLST *FILE* 'FNS).

(**FILECOMS** *FILE TYPE*) [Function]

Returns (PACK* *FILE* (OR *TYPE* 'COMS)). Note that (FILECOMS 'FOO) returns the symbol FOOCOMS, not the value of FOOCOMS.

(**SMASHFILECOMS** *FILE*) [Function]

Maps down (FILECOMSLST *FILE* 'FILEVARS) and sets to NOBIND all filevars (see the FileVars section), i.e., any variable used in a command of the form (*COMMAND* * *VARIABLE*). Also sets (FILECOMS *FILE*) to NOBIND. Returns *FILE*.

Symbolic File Format

The file manager manipulates symbolic files in a particular format. This format is defined so that the information in the file is easily readable when the file is listed, as well as being easily manipulated by the file manager functions. In general, there is no reason for you to manually change the contents of a symbolic file. However, to allow you to extend the file manager, this section describes some of the functions used to write symbolic files, and other matters related to their format.

(**PRETTYDEF** *PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES*) [Function]

Writes a symbolic file in PRETTYPRINT format for loading, using FILERDTBL as its read table. PRETTYDEF returns the name of the symbolic file that was created.

PRETTYDEF operates under a RESETLST (see Chapter 14), so if an error occurs, or a Control-D is typed, all files that PRETTYDEF has opened will be closed, the (partially complete) file being written will be deleted, and any undoable operations executed will be undone. The RESETLST also means that any RESETSAVEs executed in the file manager commands will also be protected.

PRTTYFNS is an optional list of function names. It is equivalent to including (FNS * *PRTTYFNS*) in the file manager commands in *PRTTYCOMS*. *PRTTYFNS* is an anachronism from when PRETTYDEF did not use a list of file manager commands, and should be specified as NIL.

PRTTYFILE is the name of the file on which the output is to be written. *PRTTYFILE* has to be a symbnol. If *PRTTYFILE* = NIL, the primary output file is used. *PRTTYFILE* is opened if not already open, and it becomes the primary output file. *PRTTYFILE* is closed at end of PRETTYDEF, and the primary output file is restored.

PRTTYCOMS is a list of file manager commands interpreted as described in the File Manager Commands section. If *PRTTYCOMS* is atomic, its top level value is used and an RPAQQ is written which will set that atom to the list of commands when the file is subsequently loaded. A PRETTYCOMPRINT expression (see below) will also be written which informs you of the named atom or list of commands when the file is subsequently loaded. In addition, if any of the functions in the file are nlambda functions, PRETTYDEF will automatically print a DECLARE: expression suitable for informing the compiler about these functions, in case you recompile the file without having first loaded the nlambda functions (see Chapter 18).

REPRINTFNS and *SOURCEFILE* are for use in conjunction with remaking a file (see the Remaking a Symbolic File section). *REPRINTFNS* can be a list of functions to be prettyprinted, or EXPRS, meaning prettyprint all functions with EXPR definitions, or ALL meaning prettyprint all functions either defined as EXPRs, or with EXPR properties. Note that doing a remake with *REPRINTFNS* = NIL makes sense if there have been changes in the file, but not to any of the functions, e.g., changes to variables or property lists. *SOURCEFILE* is the name of the file from which to copy the definitions for those functions that are *not* going to be prettyprinted, i.e., those not specified by *REPRINTFNS*. *SOURCEFILE* = T means to use most recent version (i.e., highest number) of

PRTTYFILE, the second argument to PRETTYDEF. If *SOURCEFILE* cannot be found, PRETTYDEF prints the message "*FILE* NOT FOUND, SO IT WILL BE WRITTEN ANEW", and proceeds as it does when *REPRINTFNS* and *SOURCEFILE* are both NIL.

PRETTYDEF calls PRETTYPRINT with its second argument *PRETTYDEFLG* = T, so whenever PRETTYPRINT starts a new function, it prints (on the terminal) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

Note that normally if PRETTYPRINT is given a symbol which is not defined as a function but is known to be on one of the files noticed by the file manager, PRETTYPRINT will load in the definition (using LOADFNS) and print it. This is not done when PRETTYPRINT is called from PRETTYDEF.

In Medley the SYSPRETTYFLG is ignored in the Interlisp exec.

(**PRINTFNS** *X*) [Function]

X is a list of functions. PRINTFNS prettyprints a DEFINEQ epression that defines the functions to the primary output stream using the primary read table. Used by PRETTYDEF to implement the FNS file manager command.

(**PRINTDATE** *FILE CHANGES*) [Function]

Prints the FILECREATED expression at beginning of PRETTYDEF files. *CHANGES* used by the file manager.

(**FILECREATED** *X*) [NLambda NoSpread Function]

Prints a message (using LISPXPRINT) followed by the time and date the file was made, which is (CAR *X*). The message is the value of PRETTYHEADER, initially "FILE CREATED". If PRETTYHEADER = NIL, nothing is printed. (CDR *X*) contains information about the file, e.g., full name, address of file map, list of changed items, etc. FILECREATED also stores the time and date the file was made on the property list of the file under the property FILEDATES and performs other initialization for the file manager.

(**PRETTYCOMPRINT** *X*) [NLambda Function]

Prints *X* (unevaluated) using LISPXPRINT, unless PRETTYHEADER = NIL.

PRETTYHEADER [Variable]

Value is the message printed by FILECREATED. PRETTYHEADER is initially "FILE CREATED". If PRETTYHEADER = NIL, neither FILECREATED nor PRETTYCOMPRINT will print anything. Thus, setting PRETTYHEADER to NIL will result in "silent loads". PRETTYHEADER is reset to NIL during greeting (see Chapter 12).

(**FILECHANGES** *FILE TYPE*) [Function]

Returns a list of the changed objects of file manager type *TYPE* from the FILECREATED expression of *FILE*. If *TYPE* = NIL, returns an alist of all of the changes, with the file manager types as the CARs of the elements..

(**FILEDATE** *FILE*) [Function]

Returns the file date contained in the FILECREATED expression of *FILE*.

(**LISPSOURCEFILEP** *FILE*) [Function]

Returns a non-NIL value if *FILE* is in file manager format and has a file map, NIL otherwise.

Copyright Notices

The system has a facility for automatically printing a copyright notice near the front of files, right after the FILECREATED expression, specifying the years it was edited and the copyright owner. The format of the copyright notice is:

```
(* Copyright (c) 1981 by Foo Bars Corporation)
```

Once a file has a copyright notice then every version will have a new copyright notice inserted into the file without your intervention. (The copyright information necessary to keep the copyright up to date is stored at the end of the file.).

Any year the file has been edited is considered a "copyright year" and therefore kept with the copyright information. For example, if a file has been edited in 1981, 1982, and 1984, then the copyright notice would look like:

```
(* Copyright (c) 1981,1982,1984 by Foo Bars Corporation)
```

When a file is made, if it has no copyright information, the system will ask you to specify the copyright owner (if COPYRIGHTFLG = T). You may specify one of the names from COPYRIGHTOWNERS, or give one of the following responses:

- Type a left-square-bracket. The system will then prompt for an arbitrary string which will be used as the owner-string
- Type a right-square-bracket, which specifies that you really do not want a copyright notice.
- Type "NONE" which specifies that this file should never have a copyright notice.

For example, if COPYRIGHTOWNERS has the value

```
((BBN "Bolt Beranek and Newman Inc.")
 (XEROX "Xerox Corporation"))
```

then for a new file FOO the following interaction will take place:

```
Do you want to Copyright FOO? Yes
Copyright owner:   (user typed ?)
one of:
BBN - Bolt Beranek and Newman Inc.
XEROX - Xerox Corporation
NONE - no copyright ever for this file
[ - new copyright owner -- type one line of text
] - no copyright notice for this file now

Copyright owner: BBN
```

Then "Foo Bars Corporation" in the above copyright notice example would have been "Bolt Beranek and Newman Inc."

The following variables control the operation of the copyright facility:

COPYRIGHTFLG [Variable]

The value of COPYRIGHTFLG determines whether copyright information is maintained in files. Its value is interpreted as follows:

NIL The system will preserve old copyright information, but will not ask you about copyrighting new files. This is the default value of COPYRIGHTFLG.

T When a file is made, if it has no copyright information, the system will ask you to specify the copyright owner.

NEVER The system will neither prompt for new copyright information nor preserve old copyright information.

DEFAULT The value of DEFAULTCOPYRIGHTOWNER (below) is used for putting copyright information in files that don't have any other copyright. The prompt "Copyright owner for file xx:" will still be printed, but the default will be filled in immediately.

COPYRIGHTOWNERS [Variable]

COPYRIGHTOWNERS is a list of entries of the form (*KEY OWNERSTRING*), where *KEY* is used as a response to ASKUSER and *OWNERSTRING* is a string which is the full identification of the owner.

DEFAULTCOPYRIGHTOWNER [Variable]

If you do not respond in DWIMWAIT seconds to the copyright query, the value of DEFAULTCOPYRIGHTOWNER is used.

Functions Used Within Source Files

The following functions are normally only used within symbolic files, to set variable values, property values, etc. Most of these have special behavior depending on file manager variables.

(**RPAQ** *VAR VALUE*) [NLambda Function]

An nlambda function like SETQ that sets the top level binding of *VAR* (unevaluated) to *VALUE*.

(**RPAQQ** *VAR VALUE*) [NLambda Function]

An nlambda function like SETQQ that sets the top level binding of *VAR* (unevaluated) to *VALUE* (unevaluated).

(**RPAQ?** *VAR VALUE*) [NLambda Function]

Similar to RPAQ, except that it does nothing if *VAR* already has a top level value other than NOBIND. Returns *VALUE* if *VAR* is reset, otherwise NIL.

RPAQ, RPAQQ, and RPAQ? generate errors if *X* is not a symbol. All are affected by the value of DFNFLG (see Chapter 10). If DFNFLG = ALLPROP (and the value of *VAR* is other than NOBIND), instead of setting *X*, the corresponding value is stored on the property list of *VAR* under the property VALUE. All are undoable.

(**ADDTOVAR** *VAR* X_1 X_2 ... X_N) [NLambda NoSpread Function]

Each X_i that is not a member of the value of *VAR* is added to it, i.e. after ADDTOVAR completes, the value of *VAR* will be (UNION (LIST X_1 X_2 ... X_N) *VAR*). ADDTOVAR is used by PRETTYDEF for implementing the ADDVARS command. It performs some file manager related operations, i.e. "notices" that *VAR* has been changed. Returns the atom *VAR* (not the value of *VAR*).

(**APPENDTOVAR** *VAR* X_1 X_2 ... X_N) [NLambda NoSpread Function]

Similar to ADDTOVAR, except that the values are added to the end tof the list, rather than at the beginning.

(**PUTPROPS** *ATM* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N) [NLambda NoSpread Function]

Nlambda nospread version of PUTPROP (none of the arguments are evaluated). For i = 1...*N*, puts property $PROP_i$, value *VALi*, on the property list of *ATM*. Performs some file manager related operations, i.e., "notices" that the corresponding properties have been changed.

(**SAVEPUT** *ATM PROP VAL*) [Function]

Same as PUTPROP, but marks the corresponding property value as having been changed (used by the file manager).

File Maps

A file map is a data structure which contains a symbolic 'map' of the contents of a file. Currently, this consists of the begin and end byte address (see GETFILEPTR, in Chapter 25) for each DEFINEQ expression in the file, the begin and end address for each function definition within the DEFINEQ, and the begin and end address for each compiled function.

MAKEFILE, PRETTYDEF, LOADFNS, RECOMPILE, and numerous other system functions depend heavily on the file map for efficient operation. For example, the file map enables LOADFNS to load selected function definitions simply by setting the file pointer to the corresponding address using SETFILEPTR, and then performing a single READ. Similarly, the file map is heavily used by the "remake" option of MAKEFILE (see the Remaking a Symbolic File section): those function definitions that have been changed since the previous version are prettyprinted; the rest are simply copied from the old file to the new one, resulting in a considerable speedup.

Whenever a file is written by MAKEFILE, a file map for the new file is built. Building the map in this case essentially comes for free, since it requires only reading the current file pointer before and after each definition is written or copied. However, building the map does require that PRETTYPRINT *know* that it is printing a DEFINEQ expression. For this reason, you should never print a DEFINEQ expression onto a file yourself, but should instead always use the FNS file manager command (see the Functions and Macros section).

The file map is stored on the property list of the root name of the file, under the property FILEMAP. In addition, MAKEFILE writes the file map on the file itself. For cosmetic reasons, the file map is written as the last expression in the file. However, the *address* of the file map in the file is (over)written into the FILECREATED expression that appears at the beginning of the file so that the file map can be rapidly accessed without having to scan the entire file. In most cases, LOAD and LOADFNS do not have to build the file map at all, since a file map will usually appear in the corresponding file, unless the file was written with BUILDMAPFLG = NIL, or was written outside of Interlisp.

Currently, file maps for *compiled* files are not written onto the files themselves. However, LOAD and LOADFNS will build maps for a compiled file when it is loaded, and store it on the property FILEMAP. Similary, LOADFNS will obtain and use the file map for a compiled file, when available.

The use and creation of file maps is controlled by the following variables:

BUILDMAPFLG [Variable]

Whenever a file is read by LOAD or LOADFNS, or written by MAKEFILE, a file map is automatically built unless BUILDMAPFLG = NIL. (BUILDMAPFLG is initially T.)

While building the map will not help the first reference to a file, it will help in future references. For example, if you perform (LOADFROM 'FOO) where FOO does not contain a file map, the LOADFROM will be (slightly) slower than if FOO did contain a file map, but subsequent calls to LOADFNS for this version of FOO will be able to use the map that was built as the result of the LOADFROM, since it will be stored on FOO's FILEMAP property.

USEMAPFLG [Variable]

If USEMAPFLG = T (the initial setting), the functions that use file maps will first check the FILEMAP property to see if a file map for this file was previously obtained or built. If not, the first expression on the file is checked to see if it is a FILECREATED expression that also contains the address of a file map. If the file map is not on the FILEMAP property or in the file, a file map will be built (unless BUILDMAPFLG = NIL).

If USEMAPFLG = NIL, the FILEMAP property and the file will not be checked for the file map. This allows you to recover in those cases where the file and its map for some reason do not agree. For example, if you use a text editor to change a symbolic file that contains a map (not recommended), inserting or deleting just one character will throw that map off. The functions which use file maps contain various integrity checks to enable them to detect that something is wrong, and to generate the error FILEMAP DOES NOT AGREE WITH CONTENTS OF *FILE*. In such cases, you can set USEMAPFLG to NIL, causing the map contained in the file to be ignored, and then reexecute the operation.

18. COMPILER

The compiler is contained in the standard Medley system. It may be used to compile functions defined in Medley, or to compile definitions stored in a file. The resulting compiled code may be stored as it is compiled, so as to be available for immediate use, or it may be written onto a file for subsequent loading.

The most common way to use the compiler is to use one of the file package functions, such as MAKEFILE (Chapter 17), which automatically updates source files, and produces compiled versions. However, it is also possible to compile individual functions defined in Medley, by directly calling the compiler using functions such as COMPILE. No matter how the compiler is called, the function COMPSET is called which asks you certain questions concerning the compilation. (COMPSET sets the free variables LAPFLG, STRF, SVFLG, LCFIL and LSTFIL which determine various modes of operation.) Those that can be answered "yes" or "no" can be answered with YES, Y, or T for "yes"; and NO, N, or NIL for "no". The questions are:

LISTING? This asks whether to generate a listing of the compiled code. The LAP and machine code are usually not of interest but can be helpful in debugging macros. Possible answers are:

- **1** Prints output of pass 1, the LAP macro code
- **2** Prints output of pass 2, the machine code
- **YES** Prints output of both passes
- **NO** Prints no listings

The variable LAPFLG is set to the answer.

FILE: This question (which only appears if the answer to LISTING? is affirmative) ask where the compiled code listing(s) should be written. Answering T will print the listings at the terminal. The variable LSTFIL is set to the answer.

REDEFINE? This question asks whether the functions compiled should be redefined to their compiled definitions. If this is answered YES, the compiled code is stored and the function definition changed, otherwise the function definition remains unchanged.

The compiler does *not* respect the value of DFNFLG (Chapter 10) when it redefines functions to their compiled definitions. Therefore, if you set DFNFLG to PROP to completely avoid inadvertantly redefining something in your running system, you *must* not answer YES to this question.

The variable STRF is set to T (if this is answered YES) or NIL.

SAVE EXPRS? This question asks whether the original defining EXPRs of functions should be saved. If answered YES, then before redefining a function to its compiled definition, the EXPR definition is saved on the property list of the function name. Otherwise they are discarded.

It is very useful to save the EXPR definitions, just in case the compiled function needs to be changed. The editing functions will retrieve this saved definition if it exists, rather than reading from a source file.

The variable SVFLG is set to T (if this is answered YES) or NIL.

OUTPUT FILE? This question asks whether (and where) the compiled definitions should be written into a file for later loading. If you answer with the name of a file, that file will be used. If you answer Y or YES, you will be asked the name of the file. If the file named is already open, it will continue to be used. If you answer T or TTY**:**, the output will be typed on the teletype (not particularly useful). If you answer N, NO, or NIL, output will *not* be done.

The variable LCFIL is set to the name of the file.

To make answering these questions easier, there are four other possible answers to the LISTING? question, which specify common compiling modes:

S **S**ame as last setting. Uses the same answers to compiler questions as given for the last compilation.

F Compile to **F**ile, without redefining functions.

ST **ST**ore new definitions, saving EXPR definitions.

STF **ST**ore new definitions; **F**orget EXPR definitions.

Implicit in these answers are the answers to the questions on disposition of compiled code and EXPR definitions, so the questions REDEFINE? and SAVE EXPRS? would not be asked if these answers were given. OUTPUT FILE? would still be asked, however. For example:

```
←COMPILE((FACT FACT1 FACT2))
LISTING? ST
OUTPUT FILE? FACT.DCOM
(FACT COMPILING)
.
.
(FACT REDEFINED)
.
.
(FACT2 REDEFINED)
(FACT FACT1 FACT2)
←
```

This process caused the functions FACT, FACT1, and FACT2 to be compiled, redefined, and the compiled definitions also written on the file FACT.DCOM for subsequent loading.

Compiler Printout

In Medley, for each function *FN* compiled, whether by TCOMPL, RECOMPILE, or COMPILE, the compiler prints:

(*FN* (*ARG*$_1$ ... *ARG*$_N$) (uses: *VAR*$_1$ ... *VAR*$_N$) (calls: *FN*$_1$ ... *FN*$_N$))

The message is printed at the beginning of the second pass of the compilation of *FN*. (*ARG*$_1$ ... *ARG*$_N$) is the list of arguments to *FN*; following uses: are the free variables referenced or set in *FN* (not including global variables); following calls: are the undefined functions called within *FN*.

If the compilation of *FN* causes the generation of one or more auxilary functions, a compiler message will be printed for these functions before the message for *FN*, e.g.,

```
(FOOA0027 (X) (uses: XX))
(FOO (A B))
```

When compiling a block, the compiler first prints (*BLKNAME* *BLKFN*$_1$ *BLKFN*$_2$ ...). Then the normal message is printed for the entire block. The names of the arguments to the block are generated by suffixing # and a number to the block name, e.g., (FOOBLOCK (FOOBLOCK#0 FOOBLOCK#1) *FREE-VARIABLES*). Then a message is printed for each *entry* to the block.

In addition to the above output, both RECOMPILE and BRECOMPILE print the name of each function that is being copied from the old compiled file to the new compiled file. The normal compiler message is printed for each function that is actually compiled.

The compiler prints out error messages when it encounters problems compiling a function. For example:

```
----- In BAZ:
***** (BAZ - illegal RETURN)
-----
```

The above error message indicates that an illegal RETURN compiler error occurred while trying to compile the function BAZ. Some compiler errors cause the compilation to terminate, producing nothing; however, there are other compiler errors which do not stop compilation. The compiler error messages are described in the last section of this chapter.

Compiler printout and error messages go to the file COUTFILE, initially T. COUTFILE can also be set to the name of a file opened for output, in which case all compiler printout will go to COUTFILE, i.e.

the compiler will compile "silently." However, any error messages will be printed to both COUTFILE as well as T.

Global Variables

Variables that appear on the list GLOBALVARS, or have the property GLOBALVAR with value T, or are declared with the GLOBALVARS file package command, are called global variables. Such variables are always accessed through their top level value when they are used freely in a compiled function. In other words, a reference to the value of a global variable is equivalent to calling GETTOPVAL on the variable, regardless of whether or not it is bound in the current access chain. Similarly, (SETQ *VARIABLE VALUE*) will compile as (SETTOPVAL (QUOTE *VARIABLE*) *VALUE*).

All system parameters, unless otherwise specified, are declared as global variables. Thus, *rebinding* these variables in a deep bound system like Medley will not affect the behavior of the system: instead, the variables must be *reset* to their new values, and if they are to be restored to their original values, reset again. For example, you might write

```
(SETQ GLOBALVARIABLE NEWVALUE)
FORM
(SETQ GLOBALVARIABLE OLDVALUE)
```

In this case, if an error occurred during the evaluation of *FORM*, or a Control-D was typed, the global variable would not be restored to its original value. The function RESETVAR provides a convenient way of resetting global variables in such a way that their values are restored even if an error occurred or Control-D is typed.

Note: The variables that a given function accesses as global variables can be determined by using the function CALLS.

Local Variables and Special Variables

In normal compiled and interpreted code, all variable bindings are accessible by lower level functions because the variable’s name is associated with its value. We call such variables *special* variables, or specvars. As mentioned earlier, the block compiler normally does *not* associate names with variable values. Such unnamed variables are not accessible from outside the function which binds them and are therefore *local* to that function. We call such unnamed variables local variables, or localvars.

The time economies of local variables can be achieved without block compiling by use of declarations. Using local variables will increase the speed of compiled code; the price is the work of writing the necessary specvar declarations for those variables which need to be accessed from outside the block.

LOCALVARS and SPECVARS are variables that affect compilation. During regular compilation, SPECVARS is normally T, and LOCALVARS is NIL or a list. This configuration causes all variables bound in the functions being compiled to be treated as special *except* those that appear on LOCALVARS. During block compilation, LOCALVARS is normally T and SPECVARS is NIL or a list. All variables are then treated as local *except* those that appear on SPECVARS.

Declarations to set LOCALVARS and SPECVARS to other values, and therefore affect how variables are treated, may be used at several levels in the compilation process with varying scope.

1. The declarations may be included in the filecoms of a file, by using the LOCALVARS and SPECVARS file package commands. The scope of the declaration is then the entire file:

```
... (LOCALVARS . T) (SPECVARS X Y) ...
```

2. The declarations may be included in block declarations; the scope is then the block, e.g.,

```
(BLOCKS ((FOOBLOCK FOO FIE (SPECVARS . T) (LOCALVARS
X)))
```

3. The declarations may also appear in individual functions, or in PROG's or LAMBDA's within a function, using the DECLARE function. In this case, the scope of the declaration is the function or the PROG or LAMBDA in which it appears. LOCALVARS and SPECVARS declarations must appear immediately after the variable list in the function, PROG, or LAMBDA, but intervening comments are permitted. For example:

```
(DEFINEQ ((FOO
     (LAMBDA (X Y)
          (DECLARE (LOCALVARS Y))
          (PROG (X Y Z)
               (DECLARE (LOCALVARS X))
               ... ]
```

If the above function is compiled (non-block), the outer X will be special, the X bound in the PROG will be local, and both bindings of Y will be local.

Declarations for LOCALVARS and SPECVARS can be used in two ways: either to cause variables to be treated the same whether the function(s) are block compiled or compiled normally, or to affect one compilation mode while not affecting the default in the other mode. For example:

```
(LAMBDA (X Y)
        (DECLARE (SPECVARS . T))
        (PROG (Z) ... ]
```

will cause X, Y, and Z to be specvars for both block and normal compilation while

```
(LAMBDA (X Y)
        (DECLARE (SPECVARS X))
        ... ]
```

will make X a specvar when block compiling, but when regular compiling the declaration will have no effect, because the default value of specvars would be T, and therefore *both* X and Y will be specvars by default.

Although LOCALVARS and SPECVARS declarations have the same form as other components of block declarations such as (LINKFNS . T), their operation is somewhat different because the two variables are not independent. (SPECVARS . T) will cause SPECVARS to be set to T, and LOCALVARS to be set to NIL. (SPECVARS V1 V2 ...) will have *no* effect if the value of SPECVARS is T, but if it is a list (or NIL), SPECVARS will be set to the union of its prior value and (V1 V2 ...). The operation of LOCALVARS is analogous. Thus, to affect both modes of compilation one of the two (LOCALVARS or SPECVARS) must be declared T before specifying a list for the other.

Note: The variables that a given function binds as local variables or accesses as special variables can be determined by using the function CALLS.

Note: LOCALVARS and SPECVARS declarations affect the compilation of local variables within a function, but the arguments to functions are always accessible as specvars. This can be changed by redefining the following function:

(**DASSEM.SAVELOCALVARS** *FN*) [Function]

This function is called by the compiler to determine whether argument information for *FN* should be written on the compiled file for *FN*. If it returns NIL, the argument information is *not* saved, and the function is stored with arguments U, V, W, etc instead of the originals.

Initially, DASSEM.SAVELOCALVARS is defined to return T. (MOVD 'NILL 'DASSEM.SAVELOCALVARS) causes the compiler to retain no local variable or argument names. Alternatively, DASSEM.SAVELOCALVARS could be redefined as a more complex predicate, to allow finer discrimination.

Constants

Interlisp allows the expression of constructions which are intended to be description of their constant values. The following functions are used to define constant values. The function SELECTC provides a mechanism for comparing a value to a number of constants.

(**CONSTANT** *X*) [Function]

This function enables you to define that the expression *X* should be treated as a "constant" value. When CONSTANT is interpreted, *X* is evaluted each time it is encountered. If the CONSTANT form is compiled, however, the expression will be evaluated only once.

If the value of *X* has a readable print name, then it will be evaluated at compile-time, and the value will be saved as a literal in the compiled function's definition, as if (QUOTE *VALUE-OF-EXPRESSION*) had appeared instead of (CONSTANT *EXPRESSION*).

If the value of *X* does not have a readable print name, then the expression *X* itself will be saved with the function, and it will be evaluated when the function is first loaded. The value will then be stored in the function's literals, and will be retrieved on future references.

If a program needed a list of 30 `NIL`s, you could specify `(CONSTANT (to 30 collect NIL))` instead of `(QUOTE (NIL NIL ...))`. The former is more concise and displays the important parameter much more directly than the latter.

`CONSTANT` can also be used to denote values that cannot be quoted directly, such as `(CONSTANT (PACK NIL))`, `(CONSTANT (ARRAY 10))`. It is also useful to parameterize quantities that are constant at run time but may differ at compile time, e.g., `(CONSTANT BITSPERWORD)` in a program is exactly equivalent to 36, if the variable `BITSPERWORD` is bound to 36 when the `CONSTANT` expression is evaluated at compile time.

Whereas the function `CONSTANT` attempts to evaluate the expression as soon as possible (compile-time, load-time, or first-run-time), other options are available, using the folowing two function:

(**LOADTIMECONSTANT** *X*) [Function]

Similar to `CONSTANT`, except that the evaluation of *X* is deferred until the compiled code for the containing function is loaded in. For example, `(LOADTIMECONSTANT (DATE))` will return the date the code was loaded. If `LOADTIMECONSTANT` is interpreted, it merely returns the value of *X*.

(**DEFERREDCONSTANT** *X*) [Function]

Similar to `CONSTANT`, except that the evaluation of *X* is always deferred until the compiled function is first run. This is useful when the storage for the constant is excessive so that it shouldn't be allocated until (unless) the function is actually invoked. If `DEFERREDCONSTANT` is interpreted, it merely returns the value of *X*.

(**CONSTANTS** VAR_1 VAR_2 ... VAR_N) [NLambda NoSpread Function]

Defines VAR_1, ... VAR_N (unevaluated) to be compile-time constants. Whenever the compiler encounters a (free) reference to one of these constants, it will compile the form (`CONSTANT` VAR_i) instead.

If VAR_i is a list of the form (*VAR FORM*), a free reference to the variable will compile as (`CONSTANT` *FORM*).

The compiler prints a warning if user code attempts to bind a variable previously declared as a constant.

Constants can be saved using the `CONSTANTS` file package command.

Compiling Function Calls

When compiling the call to a function, the compiler must know the type of the function, to determine how the arguments should be prepared (evaluated / unevaluated, spread / nospread). There are three seperate cases: lambda, nlambda spread, and nlambda nospread functions.

To determine which of these three cases is appropriate, the compiler will first look for a definition among the functions in the file that is being compiled. The function can be defined anywhere in any of the files given as arguments to BCOMPL, TCOMPL, BRECOMPILE or RECOMPILE. If the function is not contained in the file, the compiler will look for other information in the variables NLAMA, NLAML, and LAMS, which can be set by you:

NLAMA [Variable]

(For NLAMbda Atoms) A list of functions to be treated as nlambda nospread functions by the compiler.

NLAML [Variable]

(For NLAMbda List) A list of functions to be treated as nlambda spread functions by the compiler.

LAMS [Variable]

A list of functions to be treated as lambda functions by the compiler. Note that including functions on LAMS is only necessary to override in-core nlambda definitions, since in the absence of other information, the compiler assumes the function is a lambda.

If the function is not contained in a file, or on the lists NLAMA, NLAML, or LAMS, the compiler will look for a current definition in the Interlisp system, and use its type. If there is no current definition, next COMPILEUSERFN is called:

COMPILEUSERFN [Variable]

When compiling a function call, if the function type cannot be found by looking in files, the variables NLAMA, NLAML, or LAMS, or at a current definition, then if the value of COMPILEUSERFN is not NIL, the compiler calls (the value of) COMPILEUSERFN giving it as arguments CDR of the form and the form itself, i.e., the compiler does (APPLY* COMPILEUSERFN (CDR *FORM*) *FORM*). If a non-NIL value is returned, it is compiled instead of *FORM*. If NIL is returned, the compiler compiles the original expression as a call to a lambda spread that is not yet defined.

COMPILEUSERFN is only called when the compiler encounters a *list* CAR of which is not the name of a defined function. You can instruct the compiler about how to compile other data types via COMPILETYPELST.

CLISP uses COMPILEUSERFN to tell the compiler how to compile iterative statements, IF-THEN-ELSE statements, and pattern match constructs.

If the compiler cannot determine the function type by any of the means above, it assumes that the function is a lambda function, and its arguments are to be evaluated.

If there are nlambda functions called from the functions being compiled, and they are only defined in a separate file, they must be included on NLAMA or NLAML, or the compiler will incorrectly assume that their arguments are to be evaluated, and compile the calling function correspondingly. This is only necessary if the compiler does not "know" about the function. If the function is defined at compile time, or is handled via a macro, or is contained in the same group of files as the functions that call it, the compiler will automatically handle calls to that function correctly.

FUNCTION and Functional Arguments

Compiling the function FUNCTION may involve creating and compiling a seperate "auxiliary function", which will be called at run time. An auxiliary function is named by attaching a GENSYM to the end of the name of the function in which they appear, e.g., FOOA0003. For example, suppose FOO is defined as (LAMBDA (X) ... (FOO1 X (FUNCTION ...)) ...) and compiled. When FOO is run, FOO1 will be called with two arguments, X, and FOOA000*N* and FOO1 will call FOOA000*N* each time it uses its functional argument.

Compiling FUNCTION will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (MAPCAR, MAPLIST, etc.). A considerable savings in time could be achieved by making FOO1 compile open via a computed macro, e.g.

```
(PUTPROP 'FOO1 'MACRO
      '(Z (LIST (SUBST (CADADR Z)
                  (QUOTE FN)
                  DEF)
            (CAR Z)))
```

DEF is the definition of FOO1 as a function of just its first argument, and FN is the name used for its functional argument in its definition. In this case, (FOO1 X (FUNCTION ...)) would compile as an expression, containing the argument to FUNCTION as an open LAMBDA expression. Thus you save not only the function call to FOO1, but also each of the function calls to its functional argument. For example, if FOO1 operates on a list of length ten, eleven function calls will be saved. Of course, this savings in time costs space, and you must decide which is more important.

Open Functions

When a function is called from a compiled function, a system routine is invoked that sets up the parameter and control push lists as necessary for variable bindings and return information. If the amount of time spent *inside* the function is small, this function calling time will be a significant percentage of the total time required to use the function. Therefore, many "small" functions, e.g., CAR, CDR, EQ, NOT, CONS are always compiled "open", i.e., they do not result in a function call. Other larger functions such as PROG, SELECTQ, MAPC, etc. are compiled open because they are frequently used. You can make other functions compile open via MACRO definitions. You can also affect the compiled code via COMPILEUSERFN and COMPILETYPELST.

COMPILETYPELST

Most of the compiler's mechanism deals with how to handle forms (lists) and variables (symbols). You can affect the compiler's behaviour with respect to lists and literal atoms in a number of ways, e.g. macros, declarations, COMPILEUSERFN, etc. COMPILETYPELST allows you to tell the compiler what to do when it encounters a data type *other* than a list or an atom. It is the facility in the compiler that corresponds to DEFEVAL for the interpreter.

COMPILETYPELST [Variable]

A list of elements of the form (*TYPENAME* . *FUNCTION*). Whenever the compiler encounters a datum that is not a list and not an atom (or a number) in a context where the datum is being evaluated, the type name of the datum is looked up on COMPILETYPELST. If an entry appears CAR of which is equal to the type name, CDR of that entry is applied to the datum. If the value returned by this application is *not* EQ to the datum, then that value is compiled instead. If the value *is* EQ to the datum, or if there is no entry on COMPILETYPELST for this type name, the compiler simply compiles the datum as (QUOTE *DATUM*).

Compiling CLISP

Since the compiler does not know about CLISP, in order to compile functions containing CLISP constructs, the definitions must first be DWIMIFYed. You can automate this process in several ways:

1. If the variable DWIMIFYCOMPFLG is T, the compiler will always DWIMIFY expressions before compiling them. DWIMIFYCOMPFLG is initially NIL.

2. If a file has the property FILETYPE with value CLISP on its property list, TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE will operate as though DWIMIFYCOMPFLG is T and DWIMIFY all expressions before compiling.

3. If the function definition has a local CLISP declaration, including a null declaration, i.e., just (CLISP:), the definition will be automatically DWIMIFYed before compiling.

Note: COMPILEUSERFN is defined to call DWIMIFY on iterative statements, IF-THEN statements, and fetch, replace, and match expressions, i.e., any CLISP construct which can be recognized by its CAR of form. Thus, if the only CLISP constructs in a function appear inside of iterative statements, IF statements, etc., the function does not have to be dwimified before compiling.

If DWIMIFY is ever unsuccessful in processing a CLISP expression, it will print the error message UNABLE TO DWIMIFY followed by the expression, and go into a break unless DWIMESSGAG = T. In this case, the expression is just compiled as is, i.e. as though CLISP had not been enabled. You can exit the break in one of these ways:

1. Type OK to the break, which will cause the compiler to try again, e.g. you could define some missing records while in the break, and then continue

2. Type ↑, which will cause the compiler to simply compile the expression as is, i.e. as though CLISP had not been enabled in the first place

3. Return an expression to be compiled in its place by using the RETURN break command.

Note: TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE all scan the entire file before doing any compiling, and take note of the names of all functions that are defined in the file as well as the names of all variables that are set by adding them to NOFIXFNSLST and NOFIXVARSLST, respectively. Thus, if a function is not currently defined, but *is* defined in the file being compiled, when DWIMIFY is called before compiling, it will not attempt to interpret the function name as CLISP when it appears as CAR of a form. DWIMIFY also takes into account variables that have been declared to be LOCALVARS, or SPECVARS, either via block declarations or DECLARE expressions in the function being compiled, and does not attempt spelling correction on these variables. The declaration USEDFREE may also be used to declare variables simply used freely in a function. These variables will also be left alone by DWIMIFY. Finally, NOSPELLFLG is reset to T when compiling functions from a file (as opposed to from their in-core definition) so as to suppress spelling correction.

Compiler Functions

Normally, the compiler is envoked through file package commands that keep track of the state of functions, and manage a set of files, such as MAKEFILE. However, it is also possible to explicitly call the compiler using one of a number of functions. Functions may be compiled from in-core definitions (via COMPILE), or from definitions in files (TCOMPL), or from a combination of in-core and file definitions (RECOMPILE).

TCOMPL and RECOMPILE produce "compiled" files. Compiled files usually have the same name as the symbolic file they were made from, suffixed with DCOM (the compiled file extension is stored as the value of the variable COMPILE.EXT). The file name is constructed from the name field only, e.g., (TCOMPL '<BOBROW>FOO.TEM;3) produces FOO.DCOM on the connected directory. The version number will be the standard default.

A "compiled file" contains the same expressions as the original symbolic file, except for the following:

1. A special FILECREATED expression appears at the front of the file which contains information used by the file package, and which causes the message COMPILED ON *DATE* to be printed when the file is loaded (the actual string printed is the value of COMPILEHEADER).

2. Every DEFINEQ in the symbolic file is replaced by the corresponding compiled definitions in the compiled file.

3. Expressions following a DONTCOPY tag inside of a DECLARE: that appears in the symbolic file are not copied to the compiled file.

The compiled definitions appear at the front of the compiled file, i.e., before the other expressions in the symbolic file, *regardless of where they appear in the symbolic file.* The only exceptions are expressions that follow a FIRST tag inside of a DECLARE:. This "compiled" file can be loaded into any Interlisp system with LOAD.

Note: When a function is compiled from its in-core definition (as opposed to being compiled from a definition in a file), and the function has been modified by BREAK, TRACE, BREAKIN, or ADVISE, it is first restored to its original state, and a message is printed out, e.g., FOO UNBROKEN. If the function is not defined by an expr definition, the value of the function's EXPR property is used for the compilation, if there is one. If there is no EXPR property, and the compilation is being performed by RECOMPILE, the definition of the function is obtained from the file (using LOADFNS). Otherwise, the compiler prints (*FN* NOT COMPILEABLE), and goes on to the next function.

(**COMPILE** *X FLG*) [Function]

X is a list of functions (if atomic, (LIST *X*) is used). COMPILE first asks the standard compiler questions, and then compiles each function on *X*, using its in-core definition. Returns *X*.

If compiled definitions are being written to a file, the file is closed unless *FLG* = T.

(**COMPILE1** *FN DEF*) [Function]

Compiles *DEF*, redefining *FN* if STRF = T (STRF is one of the variables set by COMPSET). COMPILE1 is used by COMPILE, TCOMPL, and RECOMPILE.

If DWIMIFYCOMPFLG is T, or *DEF* contains a CLISP declaration, *DEF* is dwimified before compiling.

(**TCOMPL** *FILES*) [Function]

TCOMPL is used to "compile files"; given a symbolic LOAD file (e.g., one created by MAKEFILE), it produces a "compiled file". *FILES* is a list of symbolic files to be compiled (if atomic, (LIST *FILES*) is used). TCOMPL asks the standard compiler questions, except for "OUTPUT FILE:". The output from the compilation of each symbolic file is written on a file of the same name suffixed with DCOM, e.g., (TCOMPL '(SYM1 SYM2)) produces two files, SYM1.DCOM and SYM2.DCOM.

TCOMPL processes the files one at a time, reading in the entire file. For each FILECREATED expression, the list of functions that were marked as changed by the file package is noted, and the FILECREATED expression is written onto the output file. For each DEFINEQ expression, TCOMPL adds any nlambda functions defined in the DEFINEQ to NLAMA or NLAML, and adds lambda functions to LAMS, so that calls to these functions will be compiled correctly. NLAMA, NLAML, and LAMS are rebound to their top level values (using RESETVAR) by all of the compiling functions, so that any additions to these lists while inside of these functions will not propagate outside. Expressions beginning with DECLARE: are processed specially. All other expressions are collected to be subsequently written onto the output file.

After processing the file in this fashion, TCOMPL compiles each function, except for those functions which appear on the list DONTCOMPILEFNS (initially NIL), and writes the compiled definition onto the output file. TCOMPL then writes onto the output file the other expressions found in the symbolic file. DONTCOMPILEFNS might be used for functions that compile open, since their definitions would be superfluous when operating with the compiled file. Note that DONTCOMPILEFNS can be set via block declarations.

> Note: If the rootname of a file has the property FILETYPE with value CLISP, or value a list containing CLISP, TCOMPL rebinds DWIMIFYCOMPFLG to T while compiling the functions on *FILE*, so the compiler will DWIMIFY all expressions before compiling them.

TCOMPL returns a list of the names of the output files. All files are properly terminated and closed. If the compilation of any file is aborted via an error or Control-D, all files are properly closed, and the (partially complete) compiled file is deleted.

(**RECOMPILE** *PFILE CFILE FNS*) [Function]

The purpose of RECOMPILE is to allow you to update a compiled file without recompiling every function in the file. RECOMPILE does this by using the results of a previous compilation. It produces a compiled file similar to one that would have been produced by TCOMPL, but at a considerable savings in time by only compiling selected functions, and copying the compiled definitions for the remainder of the functions in the file from an earlier TCOMPL or RECOMPILE file.

PFILE is the name of the **P**retty file (source file) to be compiled; *CFILE* is the name of the **C**ompiled file containing compiled definitions that may be copied. *FNS* indicates which functions in *PFILE* are to be recompiled, e.g., have been changed or defined for the first time since *CFILE* was made. Note that *PFILE*, not *FNS*, drives RECOMPILE.

RECOMPILE asks the standard compiler questions, except for "OUTPUT FILE:". As with TCOMPL, the output automatically goes to *PFILE*.DCOM. RECOMPILE processes *PFILE* the same as does TCOMPL except that DEFINEQ expressions are not actually read into core. Instead, RECOMPILE uses the filemap to obtain a list of the functions contained in *PFILE*. The filemap enables RECOMPILE to skip over the DEFINEQs in the file by simply resetting the file pointer, so that in most cases the scan of the symbolic file is very fast (the only processing required is the reading of the non-DEFINEQs and the processing of the DECLARE: expressions as with TCOMPL). A map is built if the symbolic file does not already contain one, for example if it was written in an earlier system, or with BUILDMAPFLG = NIL.

After this initial scan of *PFILE*, RECOMPILE then processes the functions defined in the file. For each function in *PFILE*, RECOMPILE determines whether or not the function is to be (re)compiled. Functions that are members of DONTCOMPILEFNS are simply ignored. Otherwise, a function is recompiled if :

1. *FNS* is a list and the function is a member of that list
2. *FNS* = T or EXPRS and the function is defined by an expr definition
3. *FNS* = CHANGES and the function is marked as having been changed in the FILECREATED expression in *PFILE*
4. *FNS* = ALL

If a function is not to be recompiled, RECOMPILE obtains its compiled definition from *CFILE*, and copies it (and all generated subfunctions) to the output file, *PFILE*.DCOM. If the function does not appear on *CFILE*, RECOMPILE simply recompiles it. Finally, after processing all functions, RECOMPILE writes out all other expressions that were collected in the prescan of *PFILE*.

Note: If *FNS* = ALL, *CFILE* is superfluous, and does not have to be specified. This option may be used to compile a symbolic file that has never been compiled before, but which has already been loaded (since using TCOMPL would require reading the file in a second time).

If *CFILE* = NIL, *PFILE*.DCOM (the old version of the output file) is used for copying *from*. If both *FNS* and *CFILE* are NIL, *FNS* is set to the value of RECOMPILEDEFAULT, which is initially CHANGES. Thus you can perform his edits, dump the file, and then simply (RECOMPILE '*FILE*) to update the compiled file.

The value of RECOMPILE is the file name of the new compiled file, *PFILE*.DCOM. If RECOMPILE is aborted due to an error or Control-D, the new (partially complete) compiled file will be closed and deleted.

RECOMPILE is designed to allow you to conveniently and *efficiently* update a compiled file, even when the corresponding symbolic file has not been (completely) loaded. For example, you can perform a LOADFROM to "notice" a symbolic file, edit the functions he wants to change (the editor will automatically load those functions not already loaded), call MAKEFILE to update the symbolic file (MAKEFILE will copy the unchanged functions from the old symbolic file), and then perform (RECOMPILE *PFILE*).

Note: Since PRETTYDEF automatically outputs a suitable DECLARE: expression to indicate which functions in the file (if any) are defined as NLAMBDAs, calls to these functions will be handled correctly, even though the NLAMBDA functions themselves may never be loaded, or even looked at, by RECOMPILE.

Block Compiling

In Interlisp-10, block compiling provides a way of compiling several functions into a single block. Function calls between the component functions of the block are very fast. Thus, compiling a block consisting of just a single recursive function may be yield great savings if the function calls itself many times. The output of a block compilation is a single, usually large, function. Calls from within the block to functions outside of the block look like regular function calls. A block can be entered via several different functions, called entries. These must be specified when the block is compiled.

In Medley, block compiling is handled somewhat differently; block compiling provides a mechanism for hiding function names internal to a block, but it does not provide a performance improvement. Block compiling in Medley works by automatically renaming the block functions with special names, and calling these functions with the normal function-calling mechanisms. Specifically, a function *FN* is renamed to `\`*BLOCK-NAME*`/`*FN*. For example, function `FOO` in block `BAR` is renamed to `\BAR/FOO`. Note that it is possible with this scheme to break functions internal to a block.

Block Declarations

Block compiling a file frequently involves giving the compiler a lot of information about the nature and structure of the compilation, e.g., block functions, entries, specvars, etc. To help with this, there is the `BLOCKS` file package command, which has the form:

(BLOCKS *BLOCK*$_1$ . . . *BLOCK*$_N$)

where each *BLOCK*$_i$ is a block declaration. The `BLOCKS` command outputs a `DECLARE:` expression, which is noticed by `BCOMPL` and `BRECOMPILE`. `BCOMPL` and `BRECOMPILE` are sensitive to these declarations and take the appropriate action.

> Note: Masterscope includes a facility for checking the block declarations of a file or files for various anomalous conditions, e.g. functions in block declarations which aren't on the file(s), functions in `ENTRIES` not in the block, variables that may not need to be `SPECVARS` because they are not used freely below the places they are bound, etc.

A block declaration is a list of the form:

(*BLKNAME* *BLKFN*$_1$ . . . *BLKFN*$_M$
 (*VAR*$_1$. *VALUE*$_1$) . . . (*VAR*$_N$. *VALUE*$_N$))

BLKNAME is the name of a block. *BLKFN*$_1$ . . . *BLKFN*$_M$ are the functions in the block and correspond to *BLKFNS* in the call to `BLOCKCOMPILE`. The (*VAR*$_i$. *VALUE*$_i$) expressions indicate the settings for variables affecting the compilation of that block. If *VALUE*$_i$ is atomic, then *VAR*$_i$ is set to *VALUE*$_i$, otherwise *VAR*$_i$ is set to the `UNION` of *VALUE*$_i$ and the current value of the variable *VAR*$_i$. Also, expressions of the form (*VAR* * *FORM*) will cause *FORM* to be evaluated and the resulting list used as described above (e.g. `(GLOBALVARS * MYGLOBALVARS)`).

For example, consider the block declaration below. The block name is EDITBLOCK, it includes a number of functions (EDITL0, EDITL1, ... EDITH), and it sets the variables ENTRIES, SPECVARS, RETFNS, and GLOBALVARS.

```
(EDITBLOCK
        EDITL0 EDITL1 UNDOEDITL EDITCOM EDITCOMA
        EDITMAC EDITCOMS EDIT]UNDO UNDOEDITCOM EDITH
        (ENTRIES EDITL0 ## UNDOEDITL)
        (SPECVARS L COM LCFLG #1 #2 #3 LISPXBUFS)
        (RETFNS EDITL0)
        (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS))
```

Whenever BCOMPL or BRECOMPILE encounter a block declaration, they rebind RETFNS, SPECVARS, GLOBALVARS, BLKLIBRARY, and DONTCOMPILEFNS to their top level values, bind BLKAPPLYFNS and ENTRIES to NIL, and bind BLKNAME to the first element of the declaration. They then scan the rest of the declaration, setting these variables as described above. When the declaration is exhausted, the block compiler is called and given BLKNAME, the list of block functions, and ENTRIES.

If a function appears in a block declaration, but is not defined in one of the files, then if it has an in-core definition, this definition is used and a message printed NOT ON FILE, COMPILING IN CORE DEFINITION. Otherwise, the message NOT COMPILEABLE, is printed and the block declaration processed as though the function were not on it, i.e. calls to the function will be compiled as external function calls.

Since all compiler variables are rebound for each block declaration, the declaration only has to set those variables it wants *changed*. Furthermore, setting a variable in one declaration has no effect on the variable's value for another declaration.

After finishing all blocks, BCOMPL and BRECOMPILE treat any functions in the file that did not appear in a block declaration in the same way as do TCOMPL and RECOMPILE. If you wish a function compiled separately as well as in a block, or if you wish to compile some functions (not blockcompile), with some compiler variables changed, you can use a special pseudo-block declaration of the form

(NIL $BLKFN_1$ ... $BLKFN_M$ (VAR_1 . $VALUE_1$) ... (VAR_N . $VALUE_N$))

which means that $BLKFN_1$ ... $BLKFN_M$ should be compiled after first setting VAR_1 ... VAR_N as described above.

The following variables control other aspects of compiling a block:

RETFNS [Variable]

Value is a list of internal block functions whose names must appear on the stack, e.g., if the function is to be returned from RETFROM, RETTO, RETEVAL, etc. Usually, internal calls between functions in a block are not put on the stack.

BLKAPPLYFNS [Variable]

Value is a list of internal block functions called by other functions in the same block using BLKAPPLY or BLKAPPLY* for efficiency reasons.

Normally, a call to APPLY from inside a block would be the same as a call to any other function outside of the block. If the first argument to APPLY turned out to be one of the entries to the block, the block would have to be reentered. BLKAPPLYFNS enables a program to compute the name of a function in the block to be called next, without the overhead of leaving the block and reentering it. This is done by including on the list BLKAPPLYFNS those functions which will be called in this fashion, and by using BLKAPPLY in place of APPLY, and BLKAPPLY* in place of APPLY*. If BLKAPPLY or BLKAPPLY* is given a function not on BLKAPPLYFNS, the effect is the same as a call to APPLY or APPLY* and no error is generated. Note however, that BLKAPPLYFNS must be set at *compile* time, not run time, and furthermore, that all functions on BLKAPPLYFNS must be in the block, or an error is generated (at compile time), NOT ON BLKFNS.

BLKAPPLYFNS [Variable]

Value is a list of functions that are considered to be in the "block library" of functions that should automatically be included in the block if they are called within the block.

Compiling a function open via a macro provides a way of eliminating a function call. For block compiling, the same effect can be achieved by including the function in the block. A further advantage is that the code for this function will appear only once in the block, whereas when a function is compiled open, its code appears at each place where it is called.

The block library feature provides a convenient way of including functions in a block. It is just a convenience since you can always achieve the same effect by specifying the function(s) in question as one of the block functions, provided it has an expr definition at compile time. The block library feature simply eliminates the burden of supplying this definition.

To use the block library feature, place the names of the functions of interest on the list BLKLIBRARY, and their expr definitions on the property list of the functions under the property BLKLIBRARYDEF. When the block compiler compiles a form, it first checks to see if the function being called is one of the block functions. If not, and the function is on BLKLIBRARY, its definition is obtained from the property value of BLKLIBRARYDEF, and it is automatically included as part of the block.

Block Compiling Functions

There are three user level functions for block compiling, BLOCKCOMPILE, BCOMPL, and BRECOMPILE, corresponding to COMPILE, TCOMPL, and RECOMPILE. Note that all of the remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations, you can intermix in a single file functions compiled normally and block compiled functions.

(**BLOCKCOMPILE** *BLKNAME BLKFNS ENTRIES FLG*) [Function]

BLKNAME is the name of a block, *BLKFNS* is a list of the functions comprising the block, and *ENTRIES* a list of entries to the block.

Each of the entries must also be on *BLKFNS* or an error is generated, NOT ON BLKFNS. If only one entry is specified, the block name can also be one of the *BLKFNS*, e.g., (BLOCKCOMPILE 'FOO '(FOO FIE FUM) '(FOO)). However, if more than one entry is specified, an error will be generated, CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME.

If *ENTRIES* is NIL, (LIST *BLKNAME*) is used, e.g., (BLOCKCOMPILE 'COUNT '(COUNT COUNT1))

If *BLKFNS* is NIL, (LIST *BLKNAME*) is used, e.g., (BLOCKCOMPILE 'EQUAL)

BLOCKCOMPILE asks the standard compiler questions, and then begins compiling. As with COMPILE, if the compiled code is being written to a file, the file is closed unless *FLG* = T. The value of BLOCKCOMPILE is a list of the entries, or if *ENTRIES* = NIL, the value is *BLKNAME*.

The output of a call to BLOCKCOMPILE is one function definition for *BLKNAME*, plus definitions for each of the functions on *ENTRIES* if any. These entry functions are very short functions which immediately call *BLKNAME*.

(**BCOMPL** *FILES CFILE*) [Function]

FILES is a list of symbolic files (if atomic, (LIST *FILES*) is used). BCOMPL differs from TCOMPL in that it compiles all of the files at once, instead of one at a time, in order to permit one block to contain functions in several files. (If you have several files to be BCOMPLed *separately*, you must make several calls to BCOMPL.) Output is to *CFILE* if given, otherwise to a file whose name is (CAR *FILES*) suffixed with DCOM. For example, (BCOMPL '(EDIT WEDIT)) produces one file, EDIT.DCOM.

BCOMPL asks the standard compiler questions, except for "OUTPUT FILE:", then processes each file exactly the same as TCOMPL. BCOMPL next processes the block declarations as described above. Finally, it compiles those functions not mentioned in one of the block declarations, and then writes out all other expressions.

If *any* of the files have property FILETYPE with value CLISP, or a list containing CLISP, then DWIMIFYCOMPFLG is rebound to T for *all* of the files.

The value of BCOMPL is the output file (the new compiled file). If the compilation is aborted due to an error or Control-D, all files are closed and the (partially complete) output file is deleted.

It is permissible to TCOMPL files set up for BCOMPL; the block declarations will simply have no effect. Similarly, you can BCOMPL a file that does not contain any block declarations and the result will be the same as having TCOMPLed it.

(**BRECOMPILE** *FILES CFILE FNS* —) [Function]

BRECOMPILE plays the same role for BCOMPL that RECOMPILE plays for TCOMPL. Its purpose is to allow you to update a compiled file without requiring an entire BCOMPL.

FILES is a list of symbolic files (if atomic, (LIST *FILES*) is used). *CFILE* is the compiled file produced by BCOMPL or a previous BRECOMPILE that contains compiled definitions that may be copied. The interpretation of *FNS* is the same as with RECOMPILE.

BRECOMPILE asks the standard compiler questions, except for "OUTPUT FILE:". As with BCOMPL, output automatically goes to *FILE*.DCOM, where *FILE* is the first file in *FILES*.

BRECOMPILE processes each file the same as RECOMPILE, then processes each block declaration. If *any* of the functions in the block are to be recompiled, the entire block must be (is) recompiled. Otherwise, the block is copied from *CFILE* as with RECOMPILE. For pseudo-block declarations of the form (NIL FN_1 ...), all variable assignments are made, but only those functions indicated by *FNS* are recompiled.

After completing the block declarations, BRECOMPILE processes all functions that do not appear in a block declaration, recompiling those dictated by *FNS*, and copying the compiled definitions of the remaining from *CFILE*.

Finally, BRECOMPILE writes onto the output file the "other expressions" collected in the initial scan of *FILES*.

The value of BRECOMPILE is the output file (the new compiled file). If the compilation is aborted due to an error or Control-D, all files are closed and the (partially complete) output file is deleted.

If *CFILE* = NIL, the old version of *FILE*.DCOM is used, as with RECOMPILE. In addition, if *FNS* and *CFILE* are both NIL, *FNS* is set to the value of RECOMPILEDEFAULT, initially CHANGES.

Compiler Error Messages

Messages describing errors in the function being compiled are also printed on the terminal. These messages are always preceded by *****. Unless otherwise indicated below, the compilation will continue.

(*FN* NOT ON FILE, COMPILING IN CORE DEFINITION)

From calls to BCOMPL and BRECOMPILE.

(*FN* NOT COMPILEABLE)

An EXPR definition for *FN* could not be found. In this case, no code is produced for *FN*, and the compiler proceeds to the next function to be compiled, if any.

(*FN* NOT FOUND)

Occurs when RECOMPILE or BRECOMPILE try to copy the compiled definition of *FN* from *CFILE*, and cannot find it. In this case, no code is copied and the compiler proceeds to the next function to be compiled, if any.

(*FN* NOT ON BLKFNS)

FN was specified as an entry to a block, or else was on BLKAPPLYFNS, but did not appear on the *BLKFNS*. In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(*FN* CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME)

In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(*BLKNAME* - USED BLKAPPLY WHEN NOT APPLICABLE)

BLKAPPLY is used in the block *BLKNAME*, but there are no BLKAPPLYFNS or ENTRIES declared for the block.

(*VAR* SHOULD BE A SPECVAR - USED FREELY BY *FN*)

While compiling a block, the compiler has already generated code to bind *VAR* as a LOCALVAR, but now discovers that *FN* uses *VAR* freely. *VAR* should be declared a SPECVAR and the block recompiled.

((* --) COMMENT USED FOR VALUE)

A comment appears in a context where its value is being used, e.g. (LIST X (* --) Y). The compiled function will run, but the value at the point where the comment was used is undefined.

((*FORM*) - NON-ATOMIC CAR OF FORM)

If you intended to treat the value of *FORM* as a function, you should use APPLY* (Chapter 10). *FORM* is compiled as if APPLY* had been used.

((SETQ *VAR EXPR* --) BAD SETQ)

SETQ of more than two arguments.

(*FN* - USED AS ARG TO NUMBER FN?)

The value of a predicate, such as GREATERP or EQ, is used as an argument to a function that expects numbers, such as IPLUS.

(*FN* - NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT)

The compiler has assumed *FN* is the name of a function. If you intended to treat the *value* of *FN* as a function, APPLY* (Chapter 10) should be used. This message is printed when *FN* is not defined, and is also a local variable of the function being compiled.

(*FN* - ILLEGAL RETURN)

RETURN encountered when not in PROG.

(*TG* - ILLEGAL GO)

GO encountered when not in a PROG.

(*TG* - MULTIPLY DEFINED TAG)

TG is a PROG label that is defined more than once in a single PROG. The second definition is ignored.

(*TG* - UNDEFINED TAG)

TG is a PROG label that is referenced but not defined in a PROG.

(*VAR* - NOT A BINDABLE VARIABLE)

VAR is NIL, T, or else not a literal atom.

(*VAR VAL* -- BAD PROG BINDING)

Occurs when there is a prog binding of the form (*VAR* VAL_1 ... VAL_N).

(*TG* - MULTIPLY DEFINED TAG, LAP)

TG is a label that was encountered twice during the second pass of the compilation. If this error occurs with no indication of a multiply defined tag during pass one, the tag is in a LAP macro.

(*TG* - UNDEFINED TAG, LAP)

TG is a label that is referenced during the second pass of compilation and is not defined. LAP treats *TG* as though it were a COREVAL, and continues the compilation.

(*TG* - MULTIPLY DEFINED TAG, ASSEMBLE)

TG is a label that is defined more than once in an assemble form.

(*TG* - UNDEFINED TAG, ASSEMBLE)

TG is a label that is referenced but not defined in an assemble form.

(*OP* - OPCODE? - ASSEMBLE)

OP appears as CAR of an assemble statement, and is illegal.

(NO BINARY CODE GENERATED OR LOADED FOR *FN*)

A previous error condition was sufficiently serious that binary code for *FN* cannot be loaded without causing an error.

19. DWIM

A surprisingly large percentage of the errors made by Interlisp users are of the type that could be corrected by another Lisp programmer without any information about the purpose of the program or expression in question, e.g., misspellings, certain kinds of parentheses errors, etc. To correct these types of errors we have implemented in Medley a DWIM facility, short for Do-What-I-Mean. DWIM is called automatically whenever an error occurs in the evaluation of an Interlisp expression. (Currently, DWIM only operates on unbound atoms and undefined function errors.) DWIM then proceeds to try to correct the mistake using the current context of computation plus information about what you had previously been doing (and what mistakes you had been making) as guides to the remedy of the error. If DWIM is able to make the correction, the computation continues as though no error had occurred. Otherwise, the procedure is the same as though DWIM had not intervened: a break occurs, or an unwind to the last ERRORSET (see Chapter 14). The following protocol illustrates the operation of DWIM.

For example, suppose you define the factorial function (FACT N) as follows:

```
←DEFINEQ((FACT (LAMBDA (N) (COND
((ZEROP N0 1) ((T (ITIMS N (FACCT 9SUB1 N]
(FACT)

←
```

Note that the definition of FACT contains several mistakes: ITIMES and FACT have been misspelled; the 0 in N0 was intended to be a right parenthesis, but the Shift key was not pressed; similarly, the 9 in 9SUB1 was intended to be a left parenthesis; and finally, there is an extra left parenthesis in front of the T that begins the final clause in the conditional.

```
←PRETTYPRNT((FACCT]
       =PRETTYPRINT
       =FACT

       (FACT
         [LAMBDA (N)
           (COND
             ((ZEROP N0 1)
               ((T (ITIMS N (FACCT 9SUB1 N])
       (FACT)

←
```

After defining FACT, you want to look at its definition using PRETTYPRINT, which you unfortunately misspell. Since there is no function PRETTYPRNT in the system, an undefined function error occurs, and DWIM is called. DWIM invokes its spelling corrector, which searches a list of functions frequently used (by *this* user) for the best possible match. Finding one that is extremely close, DWIM proceeds on the assumption that PRETTYPRNT meant PRETTYPRINT, notifies you of this, and calls PRETTYPRINT.

At this point, PRETTYPRINT would normally print (FACCT NOT PRINTABLE) and exit, since FACCT has no definition. Note that this is *not* an Interlisp error condition, so that DWIM would not be called as described above. However, it is obviously not what you *meant*.

This sort of mistake is corrected by having PRETTYPRINT itself explicitly invoke the spelling corrector portion of DWIM whenever given a function with no EXPR definition. Thus, with the aid of DWIM PRETTYPRINT is able to determine that you want to see the definition of the function FACT, and proceeds accordingly.

```
←FACT(3]
       N0 [IN FACT] -> N ) ?  YES
       [IN FACT] (COND -- ((T --))) ->
                     (COND -- (T --))
       ITIMS [IN FACT] -> ITIMES
       FACCT [IN FACT] -> FACT
       9SUB1 [IN FACT] ->  ( SUB1 ?  YES
       6

←PP FACT
       (FACT
         [LAMBDA (N)
           (COND
             ((ZEROP N)
               1)
             (T (ITIMES N (FACT (SUB1 N])
       FACT

←
```

You now call FACT. During its execution, five errors occur, and DWIM is called five times. At each point, the error is corrected, a message is printed describing the action taken, and the computation is allowed to continue as if no error had occurred. Following the last correction, 6 is printed, the value of (FACT 3). Finally, you prettyprint the new, now correct, definition of FACT.

In this particular example, you were operating in TRUSTING mode, which gives DWIM carte blanche for most corrections. You can also operate in CAUTIOUS mode, in which case DWIM will inform you of intended corrections before they are made, and allow you to approve or disapprove of them. If DWIM was operating in CAUTIOUS mode in the example above, it would proceed as follows:

```
←FACT(3)
       N0 [IN FACT] -> N ) ?  YES
       U.D.F. T [IN FACT]   FIX?   YES
       [IN FACT] (COND -- ((T --))) ->
                     (COND -- (T --))
       ITIMS [IN FACT] -> ITIMES ?   ...YES
       FACCT [IN FACT] -> FACT ?   ...YES
       9SUB1 [IN FACT] ->  ( SUB1 ?  NO
       U.B.A.
       (9SUB1 BROKEN)
       :
```

For most corrections, if you do not respond in a specified interval of time, DWIM automatically proceeds with the correction, so that you need intervene only when you do not approve. In the example, you responded to the first, second, and fifth questions; DWIM responded for you on the third and fourth.

DWIM uses ASKUSER for its interactions with you (see Chapter 26). Whenever an interaction is about to take place and you have typed ahead, ASKUSER types several bells to warn you to stop typing, then clears and saves the input buffers, restoring them after the interaction is complete. Thus if you typed ahead before a DWIM interaction, DWIM will not confuse your type-ahead with the answer to its question, nor will your type-ahead be lost. The bells are printed by the function PRINTBELLS, which can be advised or redefined for specialized applications, e.g. to flash the screen for a display terminal.

A great deal of effort has gone into making DWIM "smart", and experience with a large number of users indicates that DWIM works very well; DWIM seldom fails to correct an error you feel it should have, and almost never mistakenly corrects an error. However, it is important to note that even when DWIM *is* wrong, no harm is done: since an error had occurred, you would have had to intervene anyway if DWIM took no action. Thus, if DWIM mistakenly corrects an error, you simply interrupt or abort the computation, reverse the DWIM change using UNDO (see Chapter 13), and make the correction you would have had to make without DWIM. An exception is if DWIM's correction mistakenly caused a destructive computation to be initiated, and information was lost before you could interrupt. We have not yet had such an incident occur.

(**DWIM** *X*) [Function]

Used to enable/disable DWIM. If *X* is the symbol C, DWIM is enabled in CAUTIOUS mode, so that DWIM will ask you before making corrections. If *X* is T, DWIM is enabled in TRUSTING mode, so DWIM will make most corrections automatically. If *X* is NIL, DWIM is disabled. Medley initially has DWIM enabled in CAUTIOUS mode.

DWIM returns CAUTIOUS, TRUSTING or NIL, depending to what mode it has just been put into.

For corrections to expressions typed in for immediate execution (typed into LISPX, Chapter 13), DWIM always acts as though it were in TRUSTING mode, i.e., no approval necessary. For certain types of corrections, e.g., run-on spelling corrections, 9-0 errors, etc., DWIM always acts like it was in CAUTIOUS mode, and asks for approval. In either case, DWIM always informs you of its action as described below.

Spelling Correction Protocol

One type of error that DWIM can correct is the misspelling of a function or a variable name. When an unbound symbol or undefined function error occurs, DWIM tries to correct the spelling of the bad symbol. If a symbol is found whose spelling is "close" to the offender, DWIM proceeds as follows:

If the correction occurs in the typed-in expression, DWIM prints =*CORRECT-SPELLING* and continues evaluating the expression. For example:

```
←(SETQ FOO (IPLUSS 1 2))
      =IPLUS
      3
```

If the correction does not occur in type-in, DWIM prints

BAD-SPELLING [IN *FUNCTION-NAME*] -> *CORRECT-SPELLING*

The appearance of -> is to call attention to the fact that the user's function will be or has been changed.

Then, if DWIM is in TRUSTING mode, it prints a carriage return, makes the correction, and continues the computation. If DWIM is in CAUTIOUS mode, it prints a few spaces and ? and then wait for approval. The user then has six options:

1. Type Y. DWIM types es, and proceeds with the correction.
2. Type N. DWIM types o, and does not make the correction.
3. Type ↑. DWIM does not make the correction, and furthermore guarantees that the error will not cause a break.
4. Type Control-E. For error correction, this has the same effect as typing N.
5. Do nothing. In this case DWIM waits for DWIMWAIT seconds, and if you have not responded, DWIM will type ... followed by the default answer.

   The default on spelling corrections is determined by the value of the variable FIXSPELLDEFAULT, whose top level value is initially Y.
6. Type space or carriage-return. In this case DWIM will wait indefinitely. This option is intended for those cases where you want to think about your answer, and want to insure that DWIM does not get "impatient" and answer for you.

The procedure for spelling correction on other than Interlisp errors is analogous. If the correction is being handled as type-in, DWIM prints = followed by the correct spelling, and returns it to the function that called DWIM. Otherwise, DWIM prints the incorrect spelling, followed by the correct spelling. Then, if DWIM is in TRUSTING mode, DWIM prints a carriage-return and returns the correct spelling. Otherwise, DWIM prints a few spaces and a ? and waits for approval. You can then respond with Y, N, Control-E, space, carriage return, or do nothing as described above.

The spelling corrector itself is not ERRORSET protected like the DWIM error correction routines. Therefore, typing N and typing Control-E may have different effects when the spelling corrector is called directly. The former simply instructs the spelling corrector to return NIL, and lets the calling function decide what to do next; the latter causes an error which unwinds to the last ERRORSET, however far back that may be.

Parentheses Errors Protocol

When an unbound symbol or undefined error occurs, and the offending symbol contains 9 or 0, DWIM tries to correct errors caused by typing 9 for left parenthesis and 0 for right parenthesis. In these cases, the interaction with you is similar to that for spelling correction. If the error occurs in type-in, DWIM types =*CORRECTION*, and continues evaluating the expression. For example:

```
←(SETQ FOO 9IPLUS 1 2]
       = ( IPLUS
       3
```

If the correction does not occur in type-in, DWIM prints

BAD-ATOM [IN *FUNCTION-NAME*] -> *CORRECTION* ?

and then waits for approval. You then have the same six options as for spelling correction, except the waiting time is 3*DWIMWAIT seconds. If you type Y, DWIM operates as if it were in TRUSTING mode, i.e., it makes the correction and prints its message.

Actually, DWIM uses the value of the variables LPARKEY and RPARKEY to determine the corresponding lower case character for left and right parentheses. LPARKEY and RPARKEY are initially 9 and 0 respectively, but they can be reset for other keyboard layouts, e.g., on some terminals left parenthesis is over 8, and right parenthesis is over 9.

Undefined Function T Errors

When an undefined function error occurs, and the offending function is T, DWIM tries to correct certain types of parentheses errors involving a T clause in a conditional. DWIM recognizes errors of the following forms:

(COND --) (T --)	The T clause appears outside and immediately following the COND.
(COND -- (-- & (T --)))	The T clause appears inside a previous clause.
(COND -- ((T --)))	The T clause has an extra pair of parentheses around it.

For undefined function errors that are not one of these three types, DWIM takes no corrective action at all, and the error will occur.

If the error occurs in type-in, DWIM simply types T FIXED and makes the correction. Otherwise if DWIM is in TRUSTING mode, DWIM makes the correction and prints the message:

```
[IN FUNCTION-NAME] {BAD-COND} ->
           {CORRECTED-COND}
```

If DWIM is in CAUTIOUS mode, DWIM prints

```
UNDEFINED FUNCTION T
      [IN FUNCTION-NAME]   FIX?
```

and waits for approval. You then have the same options as for spelling corrections and parenthesis errors. If you type Y or default, DWIM makes the correction and prints its message.

Having made the correction, DWIM must then decide how to proceed with the computation. In the first case, (COND --) (T --), DWIM cannot know whether the T clause would have been executed if it had been inside of the COND. Therefore DWIM asks you CONTINUE WITH T CLAUSE (with a default of YES). If you type N, DWIM continues with the form after the COND, i.e., the form that originally followed the T clause.

In the second case, (COND -- (-- & (T --))), DWIM has a different problem. After moving the T clause to its proper place, DWIM must return as the value of & as the value of the COND. Since this value is no longer around, DWIM asks you OK TO REEVALUATE and then prints the expression corresponding to &. If you type Y, or default, DWIM continues by reevaluating &, otherwise DWIM aborts, and a U.D.F. T error will then occur (even though the COND has in fact been fixed). If DWIM can determine for itself that the form can safely be reevaluated, it does not consult you before reevaluating. DWIM can do this if the form is atomic, or CAR of the form is a member of the list OKREEVALST, and each of the arguments can safely be reevaluated. For example, (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on OKREEVALST.

In the third case, (COND -- ((T --))), there is no problem with continuation, so no further interaction is necessary.

DWIM Operation

Whenever the interpreter encounters an atomic form with no binding, or a non-atomic form CAR of which is not a function or function object, it calls the function FAULTEVAL. Similarly, when APPLY is given an undefined function, FAULTAPPLY is called. When DWIM is enabled, FAULTEVAL and FAULTAPPLY are redefined to first call the DWIM package, which tries to correct the error. If DWIM cannot decide how to fix the error, or you disapprove of DWIM's correction (by typing N), or you type Control-E, then FAULTEVAL and FAULTAPPLY cause an error or break. If you type ↑ to DWIM, DWIM exits by performing (RETEVAL 'FAULTEVAL '(ERROR!)), so that an error will be generated at the position of the call to FAULTEVAL.

If DWIM can (and is allowed to) correct the error, it exits by performing RETEVAL of the corrected form, as of the position of the call to FAULTEVAL or FAULTAPPLY. Thus in the example at the beginning of the chapter, when DWIM determined that ITIMS was ITIMES misspelled, DWIM called RETEVAL with (ITIMES N (FACCT 9SUB1 N)). Since the interpreter uses the value returned by FAULTEVAL exactly as though it were the value of the erroneous form, the computation will thus proceed exactly as though no error had occurred.

In addition to continuing the computation, DWIM also repairs the cause of the error whenever possible; in the above example, DWIM also changed (with RPLACA) the expression (ITIMS N (FACCT 9SUB1 N)) that caused the error. Note that if your program had *computed* the form and called EVAL, it would not be possible to repair the cause of the error, although DWIM could correct the misspelling each time it occurred.

Error correction in DWIM is divided into three categories: unbound atoms, undefined CAR of form, and undefined function in APPLY. Assuming that the user approves DWIM's corrections, the action taken by DWIM for the various types of errors in each of these categories is summarized below.

DWIM Correction: Unbound Atoms

If DWIM is called as the result of an unbound atom error, it proceeds as follows:

1. If the first character of the unbound atom is ', DWIM assumes that you (intentionally) typed '*ATOM* for (QUOTE *ATOM*) and makes the appropriate change. No message is typed, and no approval is requested.

   If the unbound atom is just ' itself, DWIM assumes you want the *next* expression quoted, e.g., (CONS X '(A B C)) will be changed to (CONS X (QUOTE (A B C))). Again no message will be printed or approval asked. If no expression follows the ', DWIM gives up.

   > Note: ' is normally defined as a read-macro character which converts 'FOO to (QUOTE FOO) on input, so DWIM will not see the **'** in the case of expressions that are typed-in.

2. If CLISP (see Chapter 21) is enabled, and the atom is part of a CLISP construct, the CLISP transformation is performed and the result returned. For example, N-1 is transformed to (SUB1 N), and (... FOO_3 ...) is transformed into (... (SETQ FOO 3) ...).

3. If the atom contains an 9 (actually LPARKEY (see the DWIM Functions and Variables section below), DWIM assumes the **9** was intended to be a left parenthesis, and calls the editor to make appropriate repairs on the expression containing the atom. DWIM assumes that you did not notice the mistake, i.e., that the entire expression was affected by the missing left parenthesis. For example, if you type (SETQ X (LIST (CONS 9CAR Y) (CDR Z)) Y), the expression will be changed to (SETQ X (LIST (CONS (CAR Y) (CDR Z)) Y)). The 9 does not have to be the first character of the atom: DWIM will handle (CONS X9CAR Y) correctly.

4. If the atom contains a 0 (actually RPARKEY, see the DWIM Functions and Variables section below), DWIM assumes the 0 was intended to be a right parenthesis and operates as in the case above.

5. If the atom begins with a 7, the 7 is treated as a '. For example, 7FOO becomes 'FOO, and then (QUOTE FOO).

6. The expressions on DWIMUSERFORMS (see the DWIMUSERFORMS section below) are evaluated in the order that they appear. If any of these expressions returns a non-NIL value, this value is treated as the form to be used to continue the computation, it is evaluated and its value is returned by DWIM.

7. If the unbound atom occurs in a function, DWIM attempts spelling correction using the LAMBDA and PROG variables of the function as the spelling list.

8. If the unbound atom occurred in a type-in to a break, DWIM attempts spelling correction using the LAMBDA and PROG variables of the broken function as the spelling list.

9. Otherwise, DWIM attempts spelling correction using SPELLINGS3 (see the Spelling Lists section below).

10. If all of the above fail, DWIM gives up.

Undefined CAR of Form

If DWIM is called as the result of an undefined CAR of form error, it proceeds as follows:

1. If CAR of the form is T, DWIM assumes a misplaced T clause and operates as described in the Undefined Function T Errors section above.

2. If CAR of the form is F/L, DWIM changes the "F/L" to "FUNCTION(LAMBDA". For example, (F/L (Y) (PRINT (CAR Y))) is changed to (FUNCTION (LAMBDA (Y) (PRINT (CAR Y))). No message is printed and no approval requested. If you omit the variable list, DWIM supplies (X), e.g., (F/L (PRINT (CAR X))) is changed to (FUNCTION (LAMBDA (X) (PRINT (CAR X)))). DWIM determines that you have supplied the variable list when more than one expression follows F/L, CAR of the first expression is not the name of a function, and every element in the first expression is atomic. For example, DWIM will supply (X) when correcting (F/L (PRINT (CDR X)) (PRINT (CAR X))).

3. If CAR of the form is a CLISP word (IF, FOR, DO, FETCH, etc.), the indicated CLISP transformation is performed, and the result is returned as the corrected form. See Chapter 21.

4. If CAR of the form has a function definition, DWIM attempts spelling correction on CAR of the definition using as spelling list the value of LAMBDASPLST, initially (LAMBDA NLAMBDA).

5. If CAR of the form has an EXPR or CODE property, DWIM prints *CAR-OF-FORM* UNSAVED, performs an UNSAVEDEF, and continues. No approval is requested.

6. If CAR of the form has a FILEDEF property, the definition is loaded from a file (except when DWIMIFYing). If the value of the property is atomic, the entire file is to be loaded. If the value is a list, CAR is the name of the file and CDR the relevant functions, and LOADFNS will be used. For both cases, *LDFLG* will be SYSLOAD (see Chapter 17). DWIM uses FINDFILE (Chapter 24), so that the file can be on any of the directories on DIRECTORIES, initially (NIL NEWLISP LISP LISPUSERS). If the file is found, DWIM types SHALL I LOAD followed by the file name or list of functions. If you approve, DWIM loads the function(s) or file, and continues the computation.

7. If CLISP is enabled, and CAR of the form is part of a CLISP construct, the indicated transformation is performed, e.g., (N←N-1) becomes (SETQ N (SUB1 N)).

8. If CAR of the form contains an 9, DWIM assumes a left parenthesis was intended e.g., (CONS9CAR X).

9. If CAR of the form contains a 0, DWIM assumes a right parenthesis was intended.

10. If CAR of the form is a list, DWIM attempts spelling correction on CAAR of the form using LAMBDASPLST as spelling list. If successful, DWIM returns the corrected expression itself.

11. The expressions on DWIMUSERFORMS are evaluated in the order they appear. If any returns a non-NIL value, this value is treated as the corrected form, it is evaluated, and DWIM returns its value.

12. Otherwise, DWIM attempts spelling correction using SPELLINGS2 as the spelling list (see the Spelling Lists section below). When DWIMIFYing, DWIM also attemps spelling correction on function names not defined but previously encountered, using NOFIXFNSLST as a spelling list (see Chapter 21).

13. If all of the above fail, DWIM gives up.

Undefined Function in APPLY

If DWIM is called as the result of an undefined function in APPLY error, it proceeds as follows:

1. If the function has a definition, DWIM attempts spelling correction on CAR of the definition using LAMBDASPLST as spelling list.

2. If the function has an EXPR or CODE property, DWIM prints *FN* UNSAVED, performs an UNSAVEDEF and continues. No approval is requested.

3. If the function has a property FILEDEF, DWIM proceeds as in case 6 of undefined CAR of form.

4. If the error resulted from type-in, and CLISP is enabled, and the function name contains a CLISP operator, DWIM performs the indicated transformation, e.g., type FOO←(APPEND FIE FUM).

5. If the function name contains an 9, DWIM assumes a left parenthesis was intended, e.g., EDIT9FOO].

6. If the "function" is a list, DWIM attempts spelling correction on CAR of the list using LAMBDASPLST as spelling list.
7. The expressions on DWIMUSERFORMS are evaluated in the order they appear, and if any returns a non-NIL value, this value is treated as the function used to continue the computation, i.e., it will be applied to its arguments.
8. DWIM attempts spelling correction using SPELLINGS1 as the spelling list.
9. DWIM attempts spelling correction using SPELLINGS2 as the spelling list.
10. If all fail, DWIM gives up.

DWIMUSERFORMS

The variable DWIMUSERFORMS provides a convenient way of adding to the transformations that DWIM performs. For example, you might want to change atoms of the form $X to (QA4LOOKUP X). Before attempting spelling correction, but after performing other transformations (F/L, 9, 0, CLISP, etc.), DWIM evaluates the expressions on DWIMUSERFORMS in the order they appear. If any expression returns a non-NIL value, this value is treated as the transformed form to be used. If DWIM was called from FAULTEVAL, this form is evaluated and the resulting value is returned as the value of FAULTEVAL. If DWIM is called from FAULTAPPLY, this form is treated as a function to be applied to FAULTARGS, and the resulting value is returned as the value of FAULTAPPLY. If all of the expressions on DWIMUSERFORMS return NIL, DWIM proceeds as though DWIMUSERFORMS = NIL, and attempts spelling correction. Note that DWIM simply takes the value and returns it; the expressions on DWIMUSERFORMS are responsible for making any modifications to the original expression. The expressions on DWIMUSERFORMS should make the transformation permanent, either by associating it with FAULTX via CLISPTRAN, or by destructively changing FAULTX.

In order for an expression on DWIMUSERFORMS to be able to be effective, it needs to know various things about the context of the error. Therefore, several of DWIM's internal variables have been made SPECVARS (see Chapter 18) and are therefore "visible" to DWIMUSERFORMS. Below are a list of those variables that may be useful.

FAULTX [Variable]

For unbound atom and undefined car of form errors, FAULTX is the atom or form. For undefined function in APPLY errors, FAULTX is the name of the function.

FAULTARGS [Variable]

For undefined function in APPLY errors, FAULTARGS is the list of arguments. FAULTARGS may be modified or reset by expressions on DWIMUSERFORMS.

FAULTAPPLYFLG [Variable]

Value is T for undefined function in APPLY errors; NIL otherwise. The value of FAULTAPPLYFLG *after* an expression on DWIMUSERFORMS returns a non-NIL value determines how the latter value is to be treated. Following an undefined function in APPLY error, if an expression on DWIMUSERFORMS sets FAULTAPPLYFLG to NIL, the value returned is treated as a form to be evaluated, rather than a function to be applied.

FAULTAPPLYFLG is necessary to distinguish between unbound atom and undefined function in APPLY errors, since FAULTARGS may be NIL and FAULTX atomic in both cases.

TAIL [Variable]

For unbound atom errors, TAIL is the tail of the expression CAR of which is the unbound atom. DWIMUSERFORMS expression can replace the atom by another expression by performing (/RPLACA TAIL *EXPR*)

PARENT [Variable]

For unbound atom errors, PARENT is the form in which the unbound atom appears. TAIL is a tail of PARENT.

TYPE-IN? [Variable]

True if the error occurred in type-in.

FAULTFN [Variable]

Name of the function in which error occurred. FAULTFN is TYPE-IN when the error occurred in type-in, and EVAL or APPLY when the error occurred under an explicit call to EVAL or APPLY.

DWIMIFYFLG [Variable]

True if the error was encountered while DWIMIFYing (as opposed to happening while running a program).

EXPR [Variable]

Definition of FAULTFN, or argument to EVAL, i.e., the superform in which the error occurs.

The initial value of DWIMUSERFORMS is ((DWIMLOADFNS?)). DWIMLOADFNS? is a function for automatically loading functions from files. If DWIMLOADFNSFLG is T (its initial value), and CAR of the form is the name of a function, and the function is contained on a file that has been noticed by the file package, the function is loaded, and the computation continues.

DWIM Functions and Variables

DWIMWAIT [Variable]

Value is the number of seconds that DWIM will wait before it assumes that you are not going to respond to a question and uses the default response FIXSPELLDEFAULT.

DWIM operates by dismissing for 250 milliseconds, then checking to see if anything has been typed. If not, it dismisses again, etc. until DWIMWAIT seconds have elapsed. Thus, there will be a delay of at most 1/4 second before DWIM responds to your answer.

FIXSPELLDEFAULT [Variable]

If approval is requested for a spelling correction, and you do not respond, defaults to value of FIXSPELLDEFAULT, initially Y. FIXSPELLDEFAULT is rebound to N when DWIMIFYing.

ADDSPELLFLG [Variable]

If NIL, suppresses calls to ADDSPELL. Initially T.

NOSPELLFLG [Variable]

If T, suppresses *all* spelling correction. If some other non-NIL value, suppresses spelling correction in programs but not type-in. NOSPELLFLG is initially NIL. It is rebound to T when compiling from a file.

RUNONFLG [Variable]

If NIL, suppresses run-on spelling corrections. Initially NIL.

DWIMLOADFNSFLG [Variable]

If T, tells DWIM that when it encounters a call to an undefined function contained on a file that has been noticed by the file package, to simply load the function. DWIMLOADFNSFLG is initially T (see above).

LPARKEY [Variable]
RPARKEY [Variable]

DWIM uses the value of the variables LPARKEY and RPARKEY (initially 9 and 0 respectively) to determine the corresponding lower case character for left and right parentheses. LPARKEY and RPARKEY can be reset for other keyboard layouts. For example, on some terminals left parenthesis is over 8, and right parenthesis is over 9.

OKREEVALST [Variable]

The value of OKREEVALST is a list of functions that DWIM can safely reevaluate. If a form is atomic, or CAR of the form is a member of OKREEVALST, and each of the arguments can safely be reevaluated, then the form can be safely reevaluated. For example, (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on OKREEVALST.

DWIMFLG [Variable]

DWIMFLG = NIL, all DWIM operations are disabled. (DWIM 'C) and (DWIM T) set DWIMFLG to T; (DWIM NIL) sets DWIMFLG to NIL.

APPROVEFLG [Variable]

APPROVEFLG = T if DWIM should ask the user for approval before making a correction that will modify the definition of one of his functions; NIL otherwise.

When DWIM is put into CAUTIOUS mode with (DWIM 'C), APPROVEFLG is set to T; for TRUSTING mode, APPROVEFLG is set to NIL.

LAMBDASPLST [Variable]

DWIM uses the value of LAMBDASPLST as the spelling list when correcting "bad" function definitions. Initially (LAMBDA NLAMBDA). You may wish to add to LAMBDASPLST if you elect to define new "function types" via an appropriate DWIMUSERFORMS entry. For example, the QLAMBDAs of SRI's QLISP are handled in this way.

Spelling Correction

The spelling corrector is given as arguments a misspelled word (word means symbol), a spelling list (a list of words), and a number: *XWORD*, *SPLST*, and *REL* respectively. Its task is to find that word on *SPLST* which is closest to *XWORD*, in the sense described below. This word is called a *respelling* of *XWORD*. *REL* specifies the minimum "closeness" between *XWORD* and a respelling. If the spelling corrector cannot find a word on *SPLST* closer to *XWORD* than *REL*, or if it finds two or more words equally close, its value is NIL, otherwise its value is the respelling. The spelling corrector can also be given an optional functional argument, *FN*, to be used for selecting out a subset of *SPLST*, i.e., only those members of *SPLST* that satisfy *FN* will be considered as possible respellings.

The exact algorithm for computing the spelling metric is described later, but briefly "closeness" is inversely proportional to the number of disagreements between the two words, and directly proportional to the length of the longer word. For example, PRTTYPRNT is "closer" to PRETTYPRINT than CS is to CONS even though both pairs of words have the same number of disagreements. The spelling corrector operates by proceeding down *SPLST*, and computing the closeness between each word and *XWORD*, and keeping a list of those that are closest. Certain differences between words are not counted as disagreements, for example a single transposition, e.g., CONS to CNOS, or a doubled letter, e.g., CONS to CONSS, etc. In the event that the spelling corrector finds a word on *SPLST* with *no* disagreements, it will stop searching and return this word as the respelling. Otherwise, the spelling corrector continues through the entire spelling list. Then if it has found one and only one "closest" word, it returns this word as the respelling. For example, if *XWORD* is VONS, the spelling corrector will probably return CONS as the respelling. However, if *XWORD* is CONZ, the spelling corrector will not be able to return a respelling, since CONZ is equally close to both CONS and COND. If the spelling corrector finds an acceptable respelling, it interacts with you as described earlier.

In the special case that the misspelled word contains one or more $s (escape), the spelling corrector searches for those words on *SPLST* that match *XWORD*, where a $ can match any number of characters (including 0), e.g., FOO$ matches FOO1 and FOO, but not NEWFOO. FOO matches all three. Both completion and correction may be involved, e.g. RPETTY$ will match PRETTYPRINT, with one mistake. The entire spelling list is always searched, and if more than one respelling is found, the spelling corrector prints AMBIGUOUS, and returns NIL. For example, CON$ would be ambiguous if both CONS and COND were on the spelling list. If the spelling corrector finds one and only one respelling, it interacts with you as described earlier.

For both spelling correction and spelling completion, regardless of whether or not you approve of the spelling corrector's choice, the respelling is moved to the front of *SPLST*. Since many respellings are of the type with no disagreements, this procedure has the effect of considerably reducing the time required to correct the spelling of frequently misspelled words.

Synonyms

Spelling lists also provide a way of defining synonyms for a particular context. If a dotted pair appears on a spelling list (instead of just an atom), CAR is interpreted as the correct spelling of the misspelled word, and CDR as the antecedent for that word. If CAR is *identical* with the misspelled word, the antecedent is returned without any interaction or approval being necessary. If the misspelled word *corrects* to CAR of the dotted pair, the usual interaction and approval will take place, and then the antecedent, i.e., CDR of the dotted pair, is returned. For example,you could make IFLG synonymous with CLISPIFTRANFLG by adding (IFLG . CLISPIFTRANFLG) to SPELLINGS3, the spelling list for unbound atoms. Similarly, you could make OTHERWISE mean the same as ELSEIF by adding (OTHERWISE . ELSEIF) to CLISPIFWORDSPLST, or make L be synonymous with LAMBDA by adding (L . LAMBDA) to LAMBDASPLST. You can also use L as a variable without confusion, since the association of L with LAMBDA occurs only in the appropriate context.

Spelling Lists

Any list of atoms can be used as a spelling list, e.g., BROKENFNS, FILELST, etc. Various system packages have their own spellings lists, e.g., LISPXCOMS, CLISPFORWORDSPLST, EDITCOMSA, etc. These are documented under their corresponding sections, and are also indexed under "spelling lists." In addition to these spelling lists, the system maintains, i.e., automatically adds to, and occasionally prunes, four lists used solely for spelling correction: SPELLINGS1, SPELLINGS2, SPELLINGS3, and USERWORDS. These spelling lists are maintained *only* when ADDSPELLFLG is non-NIL. ADDSPELLFLG is initially T.

SPELLINGS1 [Variable]

SPELLINGS1 is a list of functions used for spelling correction when an input is typed in apply format, and the function is undefined, e.g., EDTIF(FOO). SPELLINGS1 is initialized to contain DEFINEQ, BREAK, MAKEFILE, EDITF, TCOMPL, LOAD, etc. Whenever LISPX is given an input in apply format, i.e., a function and arguments, the name of the function is added to SPELLINGS1 if the function has a definition.

For example, typing CALLS(EDITF) will cause CALLS to be added to SPELLINGS1. Thus if you typed CALLS(EDITF) and later typed CALLLS(EDITV), since SPELLINGS1 would then contain CALLS, DWIM would be successful in correcting CALLLS to CALLS.

SPELLINGS2 [Variable]

SPELLINGS2 is a list of functions used for spelling correction for all other undefined functions. It is initialized to contain functions such as ADD1, APPEND, COND, CONS, GO, LIST, NCONC, PRINT, PROG, RETURN, SETQ, etc. Whenever LISPX is given a non-atomic form, the name of the function is added to SPELLINGS2. For example, typing (RETFROM (STKPOS (QUOTE FOO) 2)) to a break would add RETFROM to SPELLINGS2. Function names are also added to SPELLINGS2 by DEFINE, DEFINEQ, LOAD (when loading compiled code), UNSAVEDEF, EDITF, and PRETTYPRINT.

SPELLINGS3 [Variable]

SPELLINGS3 is a list of words used for spelling correction on all unbound atoms. SPELLINGS3 is initialized to EDITMACROS, BREAKMACROS, BROKENFNS, and ADVISEDFNS. Whenever LISPX is given an atom to evaluate, the name of the atom is added to SPELLINGS3 if the atom has a value. Atoms are also added to SPELLINGS3 whenever they are edited by EDITV, and whenever they are set via RPAQ or RPAQQ. For example, when a file is loaded, all of the variables set in the file are added to SPELLINGS3. Atoms are also added to SPELLINGS3 when they are set by a LISPX input, e.g., typing (SETQ FOO (REVERSE (SETQ FIE ...))) will add both FOO and FIE to SPELLINGS3.

USERWORDS [Variable]

USERWORDS is a list containing both functions and variables that you have *referred* to, e.g., by breaking or editing. USERWORDS is used for spelling correction by ARGLIST, UNSAVEDEF, PRETTYPRINT, BREAK, EDITF, ADVISE, etc. USERWORDS is initially NIL. Function names are added to it by DEFINE, DEFINEQ, LOAD, (when loading compiled code, or loading exprs to property lists) UNSAVEDEF, EDITF, EDITV, EDITP, PRETTYPRINT, etc. Variable names are added to USERWORDS at the same time as they are added to SPELLINGS3. In addition, the variable LASTWORD is always set to the last word added to USERWORDS, i.e., the last function or variable referred to by the user, and the respelling of NIL is defined to be the value of LASTWORD. Thus, if you had just defined a function, you can then prettyprint it by typing PP().

Each of the above four spelling lists are divided into two sections separated by a special marker (the value of the variable SPELLSTR1). The first section contains the "permanent" words; the second section contains the temporary words. New words are added to the corresponding spelling list at the front of its temporary section (except that functions added to SPELLINGS1 or SPELLINGS2 by LISPX are always added to the end of the permanent section. If the word is already in the temporary section, it is moved to the front of that section; if the word is in the permanent section, no action is taken. If the length of the temporary section then exceeds a specified number, the last (oldest) word in the temporary section is forgotten, i.e., deleted. This procedure prevents the spelling lists from becoming cluttered with unimportant words that are no longer being used, and thereby slowing down spelling correction time. Since the spelling corrector usually moves each word selected as a respelling to the front of its spelling list, the word is thereby moved into the permanent section. Thus once a word is misspelled and corrected, it is considered important and will never be forgotten.

The spelling correction algorithm will not alter a spelling list unless it contains the special marker (the value of SPELLSTR1). This provides a way to ensure that a spelling list will not be altered.

#SPELLINGS1 [Variable]
#SPELLINGS2 [Variable]
#SPELLINGS3 [Variable]
#USERWORDS [Variable]

The maximum length of the temporary section for SPELLINGS1, SPELLINGS2, SPELLINGS3 and USERWORDS is given by the value of #SPELLINGS1, #SPELLINGS2, #SPELLINGS3, and #USERWORDS, initialized to 30, 30, 30, and 60 respectively.

You can alter these values to modify the performance behavior of spelling correction.

Generators for Spelling Correction

For some applications, it is more convenient to *generate* candidates for a respelling one by one, rather than construct a complete list of all possible candidates, e.g., spelling correction involving a large directory of files, or a natural language data base. For these purposes, *SPLST* can be an array (of any size). The first element of this array is the generator function, which is called with the array itself as its argument. Thus the function can use the remainder of the array to store "state" information, e.g., the last position on a file, a pointer into a data structure, etc. The value returned by the function is the next candidate for respelling. If NIL is returned, the spelling "list" is considered to be exhausted, and the closest match is returned. If a candidate is found with no disagreements, it is returned immediately without waiting for the "list" to exhaust.

SPLST can also be a generator, i.e. the value of the function GENERATOR (Chapter 11). The generator *SPLST* will be started up whenever the spelling corrector needs the next candidate, and it should return candidates via the function PRODUCE. For example, the following could be used as a "spelling list" which effectively contains all functions in the system:

```
[GENERATOR
      (MAPATOMS (FUNCTION (LAMBDA (X) (if (GETD X) then (PRODUCE
X]
```

Spelling Corrector Algorithm

The basic philosophy of DWIM spelling correction is to count the number of disagreements between two words, and use this number divided by the length of the longer of the two words as a measure of their relative disagreement. One minus this number is then the relative agreement or closeness. For example, CONS and CONX differ only in their last character. Such substitution errors count as one disagreement, so that the two words are in 75% agreement. Most calls to the spelling corrector specify a relative agreement of 70, so that a single substitution error is permitted in words of four characters or longer. However, spelling correction on shorter words is possible since certain types of differences such as single transpositions are not counted as disagreements. For example, AND and NAD have a relative agreement of 100. Calls to the spelling corrector from DWIM use the value of FIXSPELLREL, which is initially 70. Note that by setting FIXSPELLREL to 100, only spelling corrections with "zero" mistakes, will be considered, e.g., transpositions, double characters, etc.

The central function of the spelling corrector is CHOOZ. CHOOZ takes as arguments: a word, a minimum relative agreement, a spelling list, and an optional functional argument, *XWORD*, *REL*, *SPLST*, and *FN* respectively.

CHOOZ proceeds down *SPLST* examining each word. Words not satisfying *FN* (if *FN* is non-NIL), or those obviously too long or too short to be sufficiently close to *XWORD* are immediately rejected. For example, if *REL* = 70, and *XWORD* is 5 characters long, words longer than 7 characters will be rejected.

Special treatment is necessary for words shorter than *XWORD*, since doubled letters are not counted as disagreements. For example, CONNSSS and CONS have a relative agreement of 100. CHOOZ handles this by counting the number of doubled characters in *XWORD* before it begins scanning *SPLST*, and taking this into account when deciding whether to reject shorter words.

If *TWORD*, the current word on *SPLST*, is not rejected, CHOOZ computes the number of disagreements between it and *XWORD* by calling a subfunction, SKOR.

SKOR operates by scanning both words from left to right one character at a time. SKOR operates on the list of character codes for each word. This list is computed by CHOOZ before calling SKOR. Characters are considered to agree if they are the same characters or appear on the same key (i.e., a shift mistake). The variable SPELLCASEARRAY is a CASEARRAY which is used to determine equivalence classes for this purpose. It is initialized to equivalence lowercase and upper case letters, as well as the standard key transitions: for example, 1 with !, 3 with #, etc.

If the first character in *XWORD* and *TWORD* do *not* agree, SKOR checks to see if either character is the same as one previously encountered, and not accounted-for at that time. (In other words, transpositions are not handled by lookahead, but by *lookback*.) A displacement of two or fewer positions is counted as a tranposition; a displacement by more than two positions is counted as a disagreement.In either case, both characters are now considered as accounted for and are discarded, and SKORing continues.

If the first character in *XWORD* and *TWORD* do not agree, and neither agree with previously unaccounted-for characters, and *TWORD* has more characters remaining than *XWORD*, SKOR removes and saves the first character of *TWORD*, and continues by comparing the rest of *TWORD* with *XWORD* as described above. If *TWORD* has the same or fewer characters remaining than *XWORD*, the procedure is the same except that the character is removed from *XWORD*. In this case, a special check is first made to see if that character is equal to the *previous* character in *XWORD*, or to the *next* character in *XWORD*, i.e., a double character typo, and if so, the character is considered accounted-for, and not counted as a disagreement. In this case, the "length" of *XWORD* is also decremented. Otherwise making *XWORD* sufficiently long by adding double characters would make it be arbitrarily close to *TWORD*, e.g., XXXXXX would correct to PP.

When SKOR has finished processing both *XWORD* and *TWORD* in this fashion, the value of SKOR is the number of unaccounted-for characters, plus the number of disagreements, plus the number of tranpositions, with two qualifications:

1. If both *XWORD* and *TWORD* have a character unaccounted-for in the same position, the two characters are counted only once, i.e., substitution errors count as only one disagreement, not two
2. If there are no unaccounted-for characters and no disagreements, transpositions are not counted.

This permits spelling correction on very short words, such as edit commands, e.g., XRT->XTR. Transpositions are also not counted when FASTYPEFLG = T, for example, IPULX and IPLUS will be in 80% agreement with FASTYPEFLG = T, only 60% with FASTYPEFLG = NIL. The rationale behind this is that transpositions are much more common for fast typists, and should not be counted as disagreements, whereas more deliberate typists are not as likely to combine tranpositions and other mistakes in a single word, and therefore can use more conservative metric. FASTYPEFLG is initially NIL.

Spelling Corrector Functions and Variables

(**ADDSPELL** *X SPLST N*) [Function]

Adds *X* to one of the spelling lists as determined by the value of *SPLST*:

NIL	Adds *X* to USERWORDS and to SPELLINGS2. Used by DEFINEQ.
0	Adds *X* to USERWORDS. Used by LOAD when loading EXPRs to property lists.
1	Adds *X* to SPELLINGS1 (at end of permanent section). Used by LISPX.
2	Adds *X* to SPELLINGS2 (at end of permanent section). Used by LISPX.
3	Adds *X* to USERWORDS and SPELLINGS3.
a spelling list	If *SPLST* is a spelling list, *X* is added to it. In this case, *N* is the (optional) length of the temporary section. If *X* is already on the spelling list, and in its temporary section, ADDSPELL moves *X* to the front of that section.

ADDSPELL sets LASTWORD to *X* when *SPLST* = NIL, 0 or 3.

If *X* is not a symbol, ADDSPELL takes no action.

Note that the various systems calls to ADDSPELL, e.g., from DEFINE, EDITF, LOAD, etc., can all be suppressed by setting or binding ADDSPELLFLG to NIL (see the DWIM Functions and Variables section above).

(**MISSPELLED?** *XWORD REL SPLST FLG TAIL FN*) [Function]

If *XWORD* = NIL or $ (<esc>), MISSPELLED? prints = followed by the value of LASTWORD, and returns this as the respelling, without asking for approval. Otherwise, MISSPELLED? checks to see if *XWORD* is really misspelled, i.e., if *FN* applied to *XWORD* is true, or *XWORD* is already contained on *SPLST*. In this case, MISSPELLED? simply returns *XWORD*. Otherwise MISSPELLED? computes and returns (FIXSPELL *XWORD REL SPLST FLG TAIL FN*).

(**FIXSPELL** *XWORD REL SPLST FLG TAIL FN TIEFLG DONTMOVETOPFLG*) [Function]

The value of FIXSPELL is either the respelling of or NIL. If for some reason itself is on , then FIXSPELL aborts and calls ERROR!. If there is a possibility that is spelled correctly, MISSPELLED? should be used instead of FIXSPELL. FIXSPELL performs all of the interactions described earlier, including requesting your approval if necessary.

If *XWORD* = NIL or $ (escape), the respelling is the value of LASTWORD, and no approval is requested.

If *XWORD* contains lowercase characters, and the corresponding uppercase word is correct, i.e. on *SPLST* or satisfies *FN*, the uppercase word is returned and no interaction is performed. If FIXSPELL.UPPERCASE.QUIET is NIL (the default), a warning "=XX" is printed when coercing from "xx" to "XX". If FIXSPELL.UPPERCASE.QUIET is non-NIL, no warning is given.

If *REL* = NIL, defaults to the value of FIXSPELLREL (initially 70).

If *FLG* = NIL, the correction is handled in type-in mode, i.e., approval is never requested, and *XWORD* is not typed. If *FLG* = T, *XWORD* is typed (before the =) and approval is requested if APPROVEFLG = T. If *FLG* = NO-MESSAGE, the correction is returned with no further processing. In this case, a run-on correction will be returned as a dotted pair of the two parts of the word, and a synonym correction as a list of the form (*WORD1 WORD2*), where *WORD1* is (the corrected version of) *XWORD*, and *WORD2* is the synonym. The effect of the function CHOOZ can be obtained by calling FIXSPELL with *FLG* = NO-MESSAGE.

If *TAIL* is not NIL, and the correction is successful, CAR of *TAIL* is replaced by the respelling (using /RPLACA).

FIXSPELL will attempt to correct misspellings caused by running two words together, if the global variable RUNONFLG is non-NIL (default is NIL). In this case, approval is always requested. When a run-on error is corrected, CAR of *TAIL* is replaced by the two words, and the value of FIXSPELL is the first one. For example, if FIXSPELL is called to correct the edit command (MOVE TO AFTERCOND 3 2) with *TAIL* = (AFTERCOND 3 2), *TAIL* would be changed to (AFTER COND 2 3), and FIXSPELL would return AFTER (subject to yourapproval where necessary). If *TAIL* = T, FIXSPELL will also perform run-on corrections, returning a dotted pair of the two words in the event the correction is of this type.

If *TIEFLG* = NIL and a tie occurs, i.e., more than one word on *SPLST* is found with the same degree of "closeness", FIXSPELL returns NIL, i.e., no correction. If *TIEFLG* = PICKONE and a tie occurs, the first word is taken as the correct spelling. If *TIEFLG* = LIST, the value of FIXSPELL is a list of the respellings (even if there is only one), and FIXSPELL will not perform any interaction with you, nor modify *TAIL*, the idea being that the calling program will handle those tasks. Similarly, if *TIEFLG* = EVERYTHING, a list of all candidates whose degree of closeness is above *REL* will be returned, regardless of whether some are better than others. No interaction will be performed.

If *DONTMOVETOPFLG* = T and a correction occurs, it will *not* be moved to the front of the spelling list. Also, the spelling list will not be altered unless it contains the special marker used to separate the temporary and perminant parts of the system spelling lists (the value of SPELLSTR1).

(**FNCHECK** *FN NOERRORFLG SPELLFLG PROPFLG TAIL*) [Function]

The task of FNCHECK is to check whether *FN* is the name of a function and if not, to correct its spelling. If *FN* is the name of a function or spelling correction is successful, FNCHECK adds the (corrected) name of the function to USERWORDS using ADDSPELL, and returns it as its value.

Since FNCHECK is called by many low level functions such as ARGLIST, UNSAVEDEF, etc., spelling correction only takes place when DWIMFLG = T, so that these functions can operate in a small Interlisp system which does not contain DWIM.

NOERRORFLG informs FNCHECK whether or not the calling function wants to handle the unsuccessful case: if *NOERRORFLG* is T, FNCHECK simply returns NIL, otherwise it prints fn NOT A FUNCTION and generates a non-breaking error.

If *FN* does not have a definition, but does have an EXPR property, then spelling correction is not attempted. Instead, if *PROPFLG* = T, *FN* is considered to be the name of a function, and is returned. If *PROPFLG* = NIL, *FN* is *not* considered to be the name of a function, and NIL is returned or an error generated, depending on the value of *NOERRORFLG*.

FNCHECK calls MISSPELLED? to perform spelling correction, so that if *FN* = NIL, the value of LASTWORD will be returned. *SPELLFLG* corresponds to MISSPELLED?'s fourth argument, *FLG*. If *SPELLFLG* = T, approval will be asked if DWIM was enabled in CAUTIOUS mode, i.e., if APPROVEFLG = T. *TAIL* corresponds to the fifth argument to MISSPELLED?.

FNCHECK is currently used by ARGLIST, UNSAVEDEF, PRETTYPRINT, BREAK0, BREAKIN, ADVISE, and CALLS. For example, BREAK0 calls FNCHECK with *NOERRORFLG* = T since if FNCHECK cannot produce a function, BREAK0 wants to define a dummy one. CALLS however calls FNCHECK with *NOERRORFLG* = NIL, since it cannot operate without a function.

Many other system functions call MISSPELLED? or FIXSPELL directly. For example, BREAK1 calls FIXSPELL on unrecognized atomic inputs before attempting to evaluate them, using as a spelling list a list of all break commands. Similarly, LISPX calls FIXSPELL on atomic inputs using a list of all LISPX commands. When UNBREAK is given

the name of a function that is not broken, it calls FIXSPELL with two different spelling lists, first with BROKENFNS, and if that fails, with USERWORDS. MAKEFILE calls MISSPELLED? using FILELST as a spelling list. Finally, LOAD, BCOMPL, BRECOMPILE, TCOMPL, and RECOMPILE all call MISSPELLED? if their input file(s) won't open.

20. CLISP

The syntax of Lisp is very simple. It can be described concisely, but it makes Lisp difficult to read and write without tools. Unlike many languages, there are no reserved words in Lisp such as `IF`, `THEN`, `FOR`, `DO`, etc., nor reserved characters like `+`, `-`, `=`, `←`, etc. The only components of the language are atoms and delimiters. This eliminates the need for parsers and precedence rules, and makes Lisp programs easy to mainpuilate. For example, a Lisp interpreter can be written in one or two pages of Lisp code. This makes Lisp the most suitable programming language for writing programs that deal with other programs as data.

Human language is based on more complicated structures and relies more on special words to carry the meaning. The definiton of the factorial function looks like this in Lisp:

```
(COND ((ZEROP N) 1) (T (TIMES N (FACTORIAL ((SUB1 N))))))
```

This definition is easy to read for a machine but difficult to read for a human. CLISP is designed to make Interlisp programs easier to read and write. CLISP does this by translating various operators, conditionals, and iterative statements to Interlisp. For example, factorial can be written in CLISP:

```
(IF N = 0 THEN 1 ELSE N*(FACTORIAL N-1))
```

CLISP will translate this expression to the form in the example above. The translation will take place when the form is read so there are no performance penalties.

You should view CLISP as a shothand for produceing Lisp programs. CLISP makes a program easy to read and sometimes more compact.

CLISP is implemented via the error correction machinery in Interlisp (see Chapter 20). Any expression that Interlisp thinks is well-formed will never be seen by CLISP This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the Interlisp interpreter does not know about CLISP at all. When the interpreter finds an error it calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer. The DWIM analyzer knows how to deal with CLISP expressions. If the expression in question turns out to be a CLISP construct, the translated form is returned to the interpreter. In addition, the original CLISP expression is modified so that it *becomes* the correctly translated Interlisp form. In this way, the analysis and translation are done only once.

Integrating CLISP into Medley makes possible Do-What-I-Mean features for CLISP constructs as well as for pure Lisp expressions. For example, if you have defined a function named `GET-PARENT`, CLISP would know not to attempt to interpret the form `(GET-PARENT)` as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If you mistakenly write `(GET-PRAENT)`, CLISP would know you meant `(GET-PARENT)`, and not `(DIFFERENCE GET PRAENT)`, by using the information that `PARENT` is not the name of a variable, and that `GET-PARENT` is the name of a user function whose spelling is "very close" to that of `GET-`

PRAENT. Similarly, by using information about the program's environment not readily available to a preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

1. (LIST X*FACT N), where FACT is the name of a variable, means (LIST (X*FACT) N).
2. (LIST X*FACT N), where FACT is *not* the name of a variable but instead is the name of a function, means (LIST X*(FACT N)), i.e., N is FACT's argument.
3. (LIST X*FACT(N)), FACT the name of a function (and not the name of a variable), means (LIST X*(FACT N)).
4. Cases 1, 2 and 3 with FACT misspelled!

The first expression is correct both from the standpoint of CLISP syntax and semantics so the change would be made notification. In the other cases, you would be informed or consulted about what was taking place. For example, suppose you write the expression (LIST X*FCCT N). Assume also that there was both a function named FACT and a variable named FCT.

1. You will first be asked if FCCT is a misspelling of FCT. If you say YES, the expression will be interpreted as (LIST (X*FCT) N). If you say NO, you will be asked if FCCT was a misspelling of FACT, i.e., if you intended X*FCCT N to mean X*(FACT N).
2. If you say YES to this question, the indicated transformation will be performed. If you say NO, the system will ask if X*FCCT should be treated as CLISP, since FCCT is not the name of a (bound) variable.
3. If you say YES, the expression will be transformed, if NO, it will be left alone, i.e., as (LIST X*FCCT N). Note that we have not even considered the case where X*FCCT is itself a misspelling of a variable name, e.g., a variable named XFCT (as with GET-PRAENT). This sort of transformation will be considered after you said NO to X*FCCT N -> X*(FACT N).

The question of whether X*FCCT should be treated as CLISP is important because Interlisp users may have programs that employ identifiers containing CLISP operators. Thus, if CLISP encounters the expression A/B in a context where either A or B are not the names of variables, it will ask you if A/B is intended to be CLISP, in case you really do have a free variable named A/B.

Note: Through the discussion above, we speak of CLISP or DWIM asking you. Actually, if you typed in the expression in question for immediate execution, you are simply informed of the transformation, on the grounds that you would prefer an occasional misinterpretation rather than being continuously bothered, especially since you can always retype what you intended if a mistake occurs, and ask the programmer's assistant to UNDO the effects of the mistaken operations if necessary. For transformations on expressions in your programs, you can tell CLISP whether you wish to operate in CAUTIOUS or TRUSTING mode. In the former case (most typical) you will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing you.

CLISP can also handle parentheses errors caused by typing 8 or 9 for (or). (On most terminals, 8 and 9 are the lowercase characters for (and), i.e., (and 8 appear on the same key, as do) and 9.) For example, if you write N*8FACTORIAL N-1, the parentheses error can be detected and fixed before the infix operator * is converted to the Interlisp function TIMES. CLISP is able to distinguish this situation from cases like N*8*X meaning (TIMES N 8 X), or N*8X, where 8X is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled! For example, CLISP can successfully translate the definition of FACTORIAL:

```
(IFF N=0 THENN1 ESLE N*8FACTTORIALNN-1)
```

to the corresponding COND, while making five spelling corrections and fixing the parenthesis error. CLISP also contains a facility for converting from Interlisp back to CLISP, so that after running the above incorrect definition of FACTORIAL, you could "clispify" the now correct version to obtain (IF N=0 THEN 1 ELSE N*(FACTORIAL N-1)).

This sort of robustness prevails throughout CLISP. For example, the iterative statement permits you to say things like:

```
(FOR OLD X FROM M TO N DO (PRINT X) WHILE (PRIMEP X))
```

However, you can also write OLD (X←M), (OLD X←M), (OLD (X←M)), permute the order of the operators, e.g., (DO PRINT X TO N FOR OLD X←M WHILE PRIMEP X), omit either or both sets of parentheses, misspell any or all of the operators FOR, OLD, FROM, TO, DO, or WHILE, or leave out the word DO entirely! And, of course, you can also misspell PRINT, PRIMEP, M or N! In this example, the only thing you could not misspell is the first X, since it specifies the *name* of the variable of iteration. The other two instances of X could be misspelled.

CLISP is well integrated into Medley. For example, the above iterative statement translates into an equivalent Interlisp form using PROG, COND, GO, etc. When the interpreter subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation. Similarly, the compiler "knows" to compile the translated form. However, if you PRETTYPRINT your program, PRETTYPRINT "knows" to print the original CLISP at the corresponding point in your function. Similarly, when you edit your program, the editor keeps the translation invisible to you. If you modify the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

In short, CLISP is not a language at all, but rather a system. It plays a role analagous to that of the programmer's assistant (Chapter 13). Whereas the programmer's assistant is an invisible intermediary agent between your console requests and the Interlisp executive, CLISP sits between your programs and the Interlisp interpreter.

Only a small effort has been devoted to defining the core syntax of CLISP. Instead, most of the effort has been concentrated on providing a facility which "makes sense" out of the input expressions using context information as well as built-in and acquired information about user and system programs. It has been said that communication is based on the intention of the speaker to produce an effect in the recipient. CLISP operates under the assumption that what you say is *intended* to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide you with many different ways of saying the same thing, but to enable you to worry less about the *syntactic* aspects of your communication with the system. In other words, it gives you a new degree of freedom by permitting you to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

DWIM and CLISP are invoked on iterative statements because CAR of the iterative statement is not the name of a function, and hence generates an error. If you define a function by the same name as an i.s. operator, e.g., WHILE, TO, etc., the operator will no longer have the CLISP interpretation when it appears as CAR of a form, although it will continue to be treated as an i.s. operator if it appears in the interior of an i.s. To alert you, a warning message is printed, e.g., (WHILE DEFINED, THEREFORE DISABLED IN CLISP).

CLISP Interaction with User

Syntactically and semantically well formed CLISP transformations are always performed without informing you. Other CLISP transformations described in the previous section, e.g., misspellings of operands, infix operators, parentheses errors, unary minus - binary minus errors, all follow the same protocol as other DWIM transformations (Chapter 19). That is, if DWIM has been enabled in TRUSTING mode, or the transformation is in an expression you typed in for immediate execution, your approval is not requested, but you are informed. However, if the transformation involves a user program, and DWIM was enabled in CAUTIOUS mode, you will be asked to approve. If you say NO, the transformation is not performed. Thus, in the previous section, phrases such as "one of these (transformations) succeeds" and "the transformation LAST-ELL -> LAST-EL would be found" etc., all mean if you are in CAUTIOUS mode and the error is in a program, the corresponding transformation will be performed only if you approve (or defaults by not responding). If you say NO, the procedure followed is the same as though the transformation had not been found. For example, if A*B appears in the function FOO, and B is not bound (and no other transformations are found) you would be asked A*B [IN FOO] TREAT AS CLISP ? (The waiting time on such interactions is three times as long as for simple corrections, i.e., 3*DWIMWAIT).

In certain situations, DWIM asks for approval even if DWIM is enabled in TRUSTING mode. For example, you are always asked to approve a spelling correction that might also be interpreted as a CLISP transformation, as in LAST-ELL -> LAST-EL.

If you approved, A*B would be transformed to (ITIMES A B), which would then cause a U.B.A.B. error in the event that the program was being run (remember the entire discussion also applies to DWIMifying). If you said NO, A*B would be left alone.

If the value of CLISPHELPFLG = NIL (initally T), you will not be asked to approve any CLISP transformation. Instead, in those situations where approval would be required, the effect is the same as though you had been asked and said NO.

CLISP Character Operators

CLISP recognizes a number of special characters operators, both prefix and infix, which are translated into common expressions. For example, the character + is recognized to represent addition, so CLISP translates the symbol A+B to the form (IPLUS A B). Note that CLISP is invoked, and this translation is made, only if an error occurs, such as an unbound atom error or an undefined function error for the perfectly legitamate symbol A+B. Therefore you may choose not to use these facilities with no penalty, similar to other CLISP facilities.

You have a lot of flexability in using CLISP character operators. A list can always be substituted for a symbol, and vice versa, without changing the interpretation of a phrase. For example, if the value of (FOO X) is A, and the value of (FIE Y) is B, then (LIST (FOO X)+(FIE Y)) has the same value as (LIST A+B). Note that the first expression is a list of *four* elements: the atom "LIST", the list "(FOO X)", the atom "+", and the list "(FIE X)", whereas the second expression, (LIST A+B), is a list of only *two* elements: the symbol "LIST" and the symbol "A+B". Since (LIST (FOO X)+(FIE Y)) is indistinguishable from (LIST (FOO X) + (FIE Y)) because spaces before or after parentheses have no effect on the Interlisp READ program, to be consistent, extra spaces have no effect on atomic operands either. In other words, CLISP will treat (LIST A+ B), (LIST A +B), and (LIST A + B) the same as (LIST A+B).

Note: CLISP does not use its own special READ program because this would require you to explicitly identify CLISP expressions, instead of being able to intermix Interlisp and CLISP.

+ [CLISP Operator]
- [CLISP Operator]
***** [CLISP Operator]
/ [CLISP Operator]
↑ [CLISP Operator]

CLISP recognizes +, -, *, /, and ↑ as the normal arithmetic infix operators. The - is also recognized as the prefix operator, unary minus. These are converted to PLUS, DIFFERENCE (or in the case of unary minus, MINUS), TIMES, QUOTIENT, and EXPT.

Normally, CLISP uses the "generic" arithmetic functions PLUS, TIMES, etc. CLISP contains a facility for declaring which type of arithmetic is to be used, either by making a global declaration, or by separate declarations about individual functions or variables.

The usual precedence rules apply (although you can easily change them), i.e., * has higher precedence than + so that A+B*C is the same as A+(B*C), and both * and / are lower than ↑ so that 2*X↑2 is the same as 2*(X↑2). Operators of the same precedence group from left to right, e.g., A/B/C is equivalent to (A/B)/C. Minus is binary whenever possible, i.e., except when it is the first operator in a list, as in (-A) or (-A), or when it immediately follows another operator, as in A*-B. Note that grouping with parentheses can always be used to override the normal precedence grouping, or when you are not sure how a particular expression will parse. The complete order of precedence for CLISP operators is given below.

Note that + in front of a number will disappear when the number is read, e.g., (FOO X +2) is indistinguishable from (FOO X 2). This means that (FOO X +2) will not be interpreted as CLISP, or be converted to (FOO (IPLUS X 2)). Similarly, (FOO X -2) will not be interpreted the same as (FOO X-2). To circumvent this, always type a space between the + or - and a number if an infix operator is intended, e.g., write (FOO X + 2).

= [CLISP Operator]
GT [CLISP Operator]
LT [CLISP Operator]
GE [CLISP Operator]
LE [CLISP Operator]

These are infix operators for "Equal", "Greater Than", "Less Than", "Greater Than or Equal", and "Less Than or Equal".

GT, LT, GE, and LE are all affected by the same declarations as + and *, with the initial default to use GREATERP and LESSP.

Note that only single character operators, e.g., +, ←, =, etc., can appear in the *interior* of an atom. All other operators must be set off from identifiers with spaces. For example, XLTY will not be recognized as CLISP. In some cases, DWIM will be able to diagnose this situation as a run-on spelling error, in which case after the atom is split apart, CLISP will be able to perform the indicated transformation.

A number of Lisp functions, such as EQUAL, MEMBER, AND, OR, etc., can also be treated as CLISP infix operators. New infix operators can be easily added (see the CLISP Internal Convetions section below). Spelling correction on misspelled infix operators is peformed using CLISPINFIXSPLST as a spelling list.

AND is higher than OR, and both AND and OR are lower than the other infix operators, so (X OR Y AND Z) is the same as (X OR (Y AND Z)), and (X AND Y EQUAL Z) is the same as (X AND (Y EQUAL Z)). All of the infix predicates have lower precedence than Interlisp forms, since it is far more common to apply a predicate to two forms, than to use a Boolean as an argument to a function. Therefore, (FOO X GT FIE Y) is translated as ((FOO X) GT (FIE Y)), rather than as (FOO (X GT (FIE Y))). However, you can easily change this.

: [CLISP Operator]

X:*N* extracts the *N*th element of the list *X*. FOO:3 specifies the third element of FOO, or (CADDR FOO). If *N* is less than zero, this indicates elements counting from the end of the list; i.e. FOO:-1 is the last element of FOO. : operators can be nested, so FOO:1:2 means the second element of the first element of FOO, or (CADAR FOO).

The : operator can also be used for extracting substructures of records (see Chapter 8). Record operations are implemented by replacing expressions of the form X:FOO by (fetch FOO of X). Both lower- and uppercase are acceptable.

: is also used to indicate operations in the pattern match facility (see Chapter 12). `X:(& 'A -- 'B)` translates to (match `X` with `(& 'A -- 'B)`)

. [CLISP Operator]

In combination with :, a period can be used to specify the "data path" for record operations. For example, if `FOO` is a field of the `BAR` record, `X:BAR.FOO` is translated into (fetch (`BAR FOO`) of `X`). Subrecord fields can be specified with multiple periods: `X:BAR.FOO.BAZ` translates into (fetch `(BAR FOO BAZ)` of `X`).

Note: If a record contains fields with periods in them, `CLISPIFY` will not translate a record operation into a form using periods to specify the data path. For example, `CLISPIFY` will NOT translate `(fetch A.B of X)` into `X:A.B`.

:: [CLISP Operator]

X::*N*, returns the *N*th *tail* of the list *X*. For example, `FOO::3` is `(CDDDR FOO)`, and `FOO::-1` is `(LAST FOO)`.

← [CLISP Operator]

← is used to indicate assignment. For example, `X←Y` translates to `(SETQ X Y)`. If `X` does not have a value, and is not the name of one of the bound variables of the function in which it appears, spelling correction is attempted. However, since this may simply be a case of assigning an initial value to a new free variable, DWIM will always ask for approval before making the correction.

In conjunction with **:** and **::**, ← can also be used to perform a more general type of assignment, involving structure modification. For example, `X:2←Y` means "make the second element of `X` *be* `Y`", in Interlisp terms `(RPLACA (CDR X) Y)`. Note that the *value* of this operation is the value of `RPLACA`, which is `(CDR X)`, rather than `Y`. Negative numbers can also be used, e.g., `X:-2_Y`, which translates to `(RPLACA (NLEFT X 2) Y)`.

You can indicate you want `/RPLACA` and `/RPLACD` used (undoable version of `RPLACA` and `RPLACD`, see Chapter 13), or `FRPLACA` and `FRPLACD` (fast versions of `RPLACA` and `RPLACD`, see Chapter 3), by means of CLISP declarations. The initial default is to use `RPLACA` and `RPLACD`.

← is also used to indicate assignment in record operations (`X:FOO←Y` translates to `(replace FOO of X with Y)`.), and pattern match operations (Chapter 12).

← has different precedence on the left from on the right. On the left,← is a "tight" operator, i.e., high precedence, so that `A+B←C` is the same as `A+(B←C)`. On the right, ← has broader scope so that `A←B+C` is the same as `A←(B+C)`.

On type-in, `$←`*FORM* (where `$` is the escape key) is equivalent to set the "last thing mentioned", i.e., is equivalent to `(SET LASTWORD` *FORM*`)` (see Chapter 20). For example, immediately after examining the value of LONGVARIABLENAME, you could set it by typing $← followed by a form.

Note that an atom of the form X←Y, appearing at the top level of a PROG, will not be recognized as an assignment statement because it will be interpreted as a PROG label by the Interlisp interpreter, and therefore will not cause an error, so DWIM and CLISP will never get to see it. Instead, one must write (X←Y).

< [CLISP Operator]
> [CLISP Operator]

Angle brackets are used in CLISP to indicate list construction. The appearance of a "<" corresponds to a "(" and indicates that a list is to be constructed containing all the elements up to the corresponding ">". For example, <A B <C>> translates to (LIST A B (LIST C)). ! can be used to indicate that the next expression is to be inserted in the list as a *segment*, e.g., <A B ! C> translates to (CONS A (CONS B C)) and <! A ! B C> to (APPEND A B (LIST C)). !! is used to indicate that the next expression is to be inserted as a segment, and furthermore, all list structure to its right in the angle brackets is to be physically attached to it, e.g., <!! A B> translates to (NCONC1 A B), and <!!A !B !C> to (NCONC A (APPEND B C)). Not (NCONC (APPEND A B) C), which would have the same value, but would attach C to B, and not attach either to A. Note that <, !, !!, and > need not be separate atoms, for example, <A B ! C> may be written equally well as < A B !C >. Also, arbitrary Interlisp or CLISP forms may be used within angle brackets. For example, one can write <FOO←(FIE X) ! Y> which translates to (CONS (SETQ FOO (FIE X)) Y). CLISPIFY converts expressions in CONS, LIST, APPEND, NCONC, NCONC1, /NCONC, and /NCONC1 into equivalent CLISP expressions using <, >, !, and !!.

Note: brackets differ from other CLISP operators. For example, <A B 'C> translates to (LIST A B (QUOTE C)) even though following ', all *operators* are ignored for the rest of the identifier. (This is true only if a previous unmatched < has been seen, e.g., (PRINT 'A>B) will print the atom A>B.) Note however that <A B ' C> D> is equivalent to (LIST A B (QUOTE C>) D).

' [CLISP Operator]

CLISP recognizes ' as a prefix operator. ' means QUOTE when it is the first character in an identifier, and is ignored when it is used in the interior of an identifier. Thus, X = 'Y means (EQ X (QUOTE Y)), but X = CAN'T means (EQ X CAN'T), *not* (EQ X CAN) followed by (QUOTE T). This enables users to have variable and function names with ' in them (so long as the ' is not the first character).

Following ', all operators are ignored for the rest of the identifier, e.g., '*A means (QUOTE *A), and 'X=Y means (QUOTE X=Y), not (EQ (QUOTE X) Y). To write (EQ (QUOTE X) Y), one writes Y='X, or 'X =Y. This is one place where an extra space does make a difference.

On type-in, `'$` (escape) is equivalent to `(QUOTE` *VALUE-OF-LASTWORD*`)` (see Chapter 19). For example, after calling `PRETTYPRINT` on `LONGFUNCTION`, you could move its definition to `FOO` by typing `(MOVD '$ 'FOO)`.

Note that this is not `(MOVD $ 'FOO)`, which would be equivalent to `(MOVD LONGFUNCTION 'FOO)`, and would (probably) cause a `U.B.A. LONGFUNCTION` error, nor `MOVD($ FOO)`, which would actually move the definition of `$` to `FOO`, since DWIM and the spelling corrector would never be invoked.

~ [CLISP Operator]

CLISP recognizes `~` as a prefix operator meaning `NOT`. `~` can negate a form, as in `~(ASSOC X Y)`, or `~X`, or negate an infix operator, e.g., `(A ~GT B)` is the same as `(A LEQ B)`. Note that `~A = B` means `(EQ (NOT A) B)`.

When `~` negates an operator, e.g., `~=`, `~LT`, the two operators are treated as a single operator whose precedence is that of the second operator. When `~` negates a function, e.g., `(~FOO X Y)`, it negates the whole form, i.e., `(~(FOO X Y))`.

Order of Precedence of CLISP Operators:

`'`
`:`
← (left precedence)
`-` (unary), `~`
↑
`*, /`
`+, -` (binary)
← (right precedence)
`=`

Interlisp forms

`LT, GT, EQUAL, MEMBER`, etc.
`AND`
`OR`
`IF, THEN, ELSEIF, ELSE`
iterative statement operators

Declarations

CLISP declarations are used to affect the choice of Interlisp function used as the translation of a particular operator. For example, `A+B` can be translated as either `(PLUS A B)`, `(FPLUS A B)`, or `(IPLUS A B)`, depending on the declaration in effect. Similarly `X:1←Y` can mean `(RPLACA X Y)`, `(FRPLACA X Y)`, or `(/RPLACA X Y)`, and `<!! A B>` either `(NCONC1 A B)` or `(/NCONC1 A B)`. Note that the choice of function on all CLISP transformations are affected by the CLISP declaration in effect, i.e., iterative statements, pattern matches, record operations, as well as infix and prefix operators.

(**CLISPDEC** *DECLST*) [Function]

Puts into effect the declarations in *DECLST*. CLISPDEC performs spelling corrections on words not recognized as declarations. CLISPDEC is undoable.

You can makes (changes) a global declaration by calling CLISPDEC with *DECLST* a list of declarations, e.g., (CLISPDEC '(FLOATING UNDOABLE)). Changing a global declaration does not affect the speed of subsequent CLISP transformations, since all CLISP transformation are table driven (i.e., property list), and global declarations are accomplished by making the appropriate internal changes to CLISP at the time of the declaration. If a function employs *local* declarations (described below), there will be a slight loss in efficiency owing to the fact that for each CLISP transformation, the declaration list must be searched for possibly relevant declarations.

Declarations are implemented in the order that they are given, so that later declarations override earlier ones. For example, the declaration FAST specifies that FRPLACA, FRPLACD, FMEMB, and FLAST be used in place of RPLACA, RPLACD, MEMB, and LAST; the declaration RPLACA specifies that RPLACA be used. Therefore, the declarations (FAST RPLACA RPLACD) will cause FMEMB, FLAST, RPLACA, and RPLACD to be used.

The initial global declaration is MIXED and STANDARD.

The table below gives the declarations available in CLISP, and the Interlisp functions they indicate:

Declaration:	Interlisp Functions to be used:
MIXED	PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, LESSP, GREATERP
INTEGER or FIXED	IPLUS, IMINUS, IDIFFERENCE, ITIMES, IQUOTIENT, ILESSP, IGREATERP
FLOATING	FPLUS, FMINUS, FDIFFERENCE, FTIMES, FQUOTIENT, LESSP, FGREATERP
FAST	FRPLACA, FRPLACD, FMEMB, FLAST, FASSOC
UNDOABLE	/RPLACA, /RPLACD, /NCONC, /NCONC1, /MAPCONC, /MAPCON
STANDARD	RPLACA, RPLACD, MEMB, LAST, ASSOC, NCONC, NCONC1, MAPCONC, MAPCON
RPLACA, RPLACD, /RPLACA, etc.	corresponding function

You can also make local declarations affecting a selected function or functions by inserting an expression of the form (CLISP: . *DECLARATIONS*) immediately following the argument list, i.e., as CADDR of the definition. Such local declarations take precedence over global declarations. Declarations affecting selected variables can be indicated by lists, where the first element is the name of a variable, and the rest of the list the declarations for that variable. For example, (CLISP: FLOATING (X INTEGER)) specifies that in this function integer arithmetic be used for computations involving X, and floating arithmetic for all other computations, where "involving" means where the variable itself is an operand. For example, with the declaration (FLOATING (X INTEGER)) in effect, (FOO X)+(FIE X) would translate to FPLUS, i.e., use floating arithmetic, even though X appears somewhere inside of the operands, whereas X+(FIE X) would translate to IPLUS. If there are declarations involving *both* operands, e.g., X+Y, with (X FLOATING) (Y INTEGER), whichever appears first in the declaration list will be used.

You can also make local record declarations by inserting a record declaration, e.g., (RECORD --), (ARRAYRECORD --), etc., in the local declaration list. In addition, a local declaration of the form (RECORDS A B C) is equivalent to having copies of the global declarations A, B, and C in the local declaration. Local record declarations override global record declarations for the function in which they appear. Local declarations can also be used to override the global setting of certain DWIM/CLISP parameters effective only for transformations within that function, by including in the local declaration an expression of the form (*VARIABLE* = *VALUE*), e.g., (PATVARDEFAULT = QUOTE).

The CLISP: expression is converted to a comment of a special form recognized by CLISP. Whenever a CLISP transformation that is affected by declarations is about to be performed in a function, this comment will be searched for a relevant declaration, and if one is found, the corresponding function will be used. Otherwise, if none are found, the global declaration(s) currently in effect will be used.

Local declarations are effective in the order that they are given, so that later declarations can be used to override earlier ones, e.g., (CLISP: FAST RPLACA RPLACD) specifies that FMEMB, FLAST, RPLACA, and RPLACD be used. An exception to this is that declarations for specific variables take precedence of general, function-wide declarations, regardless of the order of appearance, as in (CLISP: (X INTEGER) FLOATING).

CLISPIFY also checks the declarations in effect before selecting an infix operator to ensure that the corresponding CLISP construct would in fact translate back to this form. For example, if a FLOATING declaration is in effect, CLISPIFY will convert (FPLUS X Y) to X+Y, but leave (IPLUS X Y) as is. If (FPLUS X Y) is CLISPIFYed while a FLOATING declaration is under effect, and then the declaration is changed to INTEGER, when X+Y is translated back to Interlisp, it will become (IPLUS X Y).

CLISP Operation

CLISP is a part of the basic Medley system. Without any special preparations, you can include CLISP constructs in programs, or type them in directly for evaluation (in EVAL or APPLY format), then, when the "error" occurrs, and DWIM is called, it will destructively transform the CLISP to the equivalent Interlisp expression and evaluate the Interlisp expression. CLISP transformations, like all DWIM corrections, are undoable. User approval is not requested, and no message is printed. This entire discussion also applies to CLISP transformation initiated by calls to DWIM from DWIMIFY.

However, if a CLISP construct contains an error, an appropriate diagnostic is generated, and the form is left unchanged. For example, if you write (LIST X+Y*), the error diagnostic MISSING OPERAND AT X+Y* IN (LIST X+Y*) would be generated. Similarly, if you write (LAST+EL X), CLISP knows that ((IPLUS LAST EL) X) is not a valid Interlisp expression, so the error diagnostic MISSING OPERATOR IN (LAST+EL X) is generated. (For example, you might have meant to say (LAST+EL*X).) If LAST+EL were the name of a defined function, CLISP would never see this form.

Since the bad CLISP transformation might not be CLISP at all, for example, it might be a misspelling of a user function or variable, DWIM holds all CLISP error messages until after trying other corrections. If one of these succeeds, the CLISP message is discarded. Otherwise, if all fail, the message is printed (but no change is made). For example, suppose you type (R/PLACA X Y). CLISP generates a diagnostic, since ((IQUOTIENT R PLACA) X Y) is obviously not right. However, since R/PLACA spelling corrects to /RPLACA, this diagnostic is never printed.

Note: CLISP error messages are not printed on type-in. For example, typing X+*Y will just produce a U.B.A. X+*Y message.

If a CLISP infix construct is well formed from a syntactic standpoint, but one or both of its operands are atomic and not bound, it is possible that either the operand is misspelled, e.g., you wrote X+YY for X+Y, or that a CLISP transformation operation was not intended at all, but that the entire expression is a misspelling. For the purpose of DWIMIFYing, "not bound" means no top level value, not on list of bound variables built up by DWIMIFY during its analysis of the expression, and not on NOFIXVARSLST, i.e., not previously seen.

For example, if you have a variable named LAST-EL, and write (LIST LAST-ELL). Therefore, CLISP computes, but does not actually perform, the indicated infix transformation. DWIM then continues, and if it is able to make another correction, does so, and ignores the CLISP interpretation. For example, with LAST-ELL, the transformation LAST-ELL -> LAST-EL would be found.

If no other transformation is found, and DWIM is about to interpret a construct as CLISP for which one of the operands is not bound, DWIM will ask you whether CLISP was intended, in this case by printing LAST-ELL TREAT AS CLISP ?.

Note: If more than one infix operator was involved in the CLISP construct, e.g., X+Y+Z, or the operation was an assignment to a variable already noticed, or TREATASCLISPFLG is T (initially NIL), you will simply be informed of the correction, e.g., X+Y+Z TREATED AS CLISP. Otherwise, even if DWIM was enabled in TRUSTING mode, you will be asked to approve the correction.

The same sort of procedure is followed with 8 and 9 errors. For example, suppose you write FOO8*X where FOO8 is not bound. The CLISP transformation is noted, and DWIM proceeds. It next asks you to approve FOO8*X -> FOO (*X. For example, this would make sense if you have (or plan to define) a function named *X. If you refuses, you are asked whether FOO8*X is to be treated as CLISP.

Similarly, if FOO8 were the name of a variable, and you write FOOO8*X, you will first be asked to approve FOOO8*X -> FOOO (XX, and if you refuse, then be offered the FOOO8 -> FOO8 correction. The 8-9 transformation is tried before spelling correction since it is empirically more likely that an unbound atom or undefined function containing an 8 or a 9 is a parenthesis error, rather than a spelling error.

CLISP also contains provision for correcting misspellings of infix operators (other than single characters), IF words, and i.s. operators. This is implemented in such a way that the user who does not misspell them is not penalized. For example, if you write IF N = 0 THEN 1 ELSSE N*(FACT N-1) CLISP does *not* operate by checking each word to see if it is a misspelling of IF, THEN, ELSE, or ELSEIF, since this would seriously degrade CLISP's performance on *all* IF statements. Instead, CLISP assumes that all of the IF words are spelled correctly, and transforms the expression to (COND ((ZEROP N) 1 ELSSE N*(FACT N-1))). Later, after DWIM cannot find any other interpretation for ELSSE, and using the fact that this atom originally appeared in an IF statement, DWIM attempts spelling correction, using (IF THEN ELSE ELSEIF) for a spelling list. When this is successful, DWIM "fails" all the way back to the original IF statement, changes ELSSE to ELSE, and starts over. Misspellings of AND, OR, LT, GT, etc. are handled similarly.

CLISP also contains many Do-What-I-Mean features besides spelling corrections. For example, the form (LIST +X Y) would generate a MISSING OPERATOR error. However, (LIST -X Y) makes sense, if the minus is unary, so DWIM offers this interpretation to you. Another common error, especially for new users, is to write (LIST X*FOO(Y)) or (LIST X*FOO Y), where FOO is the name of a function, instead of (LIST X*(FOO Y)). Therefore, whenever an operand that is not bound is also the name of a function (or corrects to one), the above interpretations are offered.

CLISP Translations

The translation of CLISP character operators and the CLISP word IF are handled by *replacing* the CLISP expression with the corresponding Interlisp expression, and discarding the original CLISP. This is done because (1) the CLISP expression is easily recomputable (by CLISPIFY) and (2) the Interlisp expressions are simple and straightforward. Another reason for discarding the original CLISP is that it may contain errors that were corrected in the course of translation (e.g., FOO←FOOO:1, N*8FOO X), etc.). If the original CLISP were retained, either you would have to go back and fix these errors by hand, thereby negating the advantage of having DWIM perform these corrections, or else DWIM would have to keep correcting these errors over and over.

Note that CLISPIFY is sufficiently fast that it is practical for you to configure your Interlisp system so that all expressions are automatically CLISPIFYed immediately before they are presented to you. For example, you can define an edit macro to use in place of P which calls CLISPIFY on the current expression before printing it. Similarly, you can inform PRETTYPRINT to call CLISPIFY on each expression before printing it, etc.

Where (1) or (2) are not the case, e.g., with iterative statements, pattern matches, record expressions, etc. the original CLISP *is* retained (or a slightly modified version thereof), and the translation is stored elsewhere (by the function CLISPTRAN, in the Miscellaneous Functions and Variables), usually in the hash array CLISPARRAY. The interpreter automatically checks this array when given a form CAR of which is not a function. Similarly, the compiler performs a GETHASH when given a form it does not recognize to see if it has a translation, which is then compiled instead of the form. Whenever you *change* a CLISP expresson by editing it, the editor automatically deletes its translation (if one exists), so that the next time it is evaluated or DWIMIFIed, the expression will be retranslated (if the value of CLISPRETRANFLG is T, DWIMIFY will also (re)translate any expressions which have translations stored remotely, see the CLISPIFY section). The function PPT and the edit commands PPT and CLISP: are available for examining translations (see the Miscellaneous Functions and Variables section).

You can also indicate that you want the original CLISP retained by embedding it in an expression of the form (CLISP . *CLISP-EXPRESSION*), e.g., (CLISP X:5:3) or (CLISP <A B C ! D>). In such cases, the translation will be stored remotely as described above. Furthermore, such expressions will be treated as CLISP even if infix and prefix transformations have been disabled by setting CLISPFLG to NIL (see the Miscellaneous Functions and Variables section). In other words, you can instruct the system to interpret as CLISP infix or prefix constructs only those expressions that are specifically flagged as such. You can also include CLISP declarations by writing (CLISP *DECLARATIONS* . *FORM*), e.g., (CLISP (CLISP: FLOATING) ...). These declarations will be used in place of any CLISP declarations in the function definition. This feature provides a way of including CLISP declarations in macro definitions.

Note: CLISP translations can also be used to supply an interpretation for function objects, as well as forms, either for function objects that are used openly, i.e., appearing as CAR of form, function objects that are explicitly APPLYed, as with arguments to mapping functions, or function objects contained in function definition cells. In all cases, if CAR of the object is not LAMBDA or NLAMBDA, the interpreter and compiler will check CLISPARRAY.

DWIMIFY

DWIMIFY is effectively a preprocessor for CLISP. DWIMIFY operates by scanning an expression as though it were being interpreted, and for each form that would generate an error, calling DWIM to "fix" it. DWIMIFY performs *all* DWIM transformations, not just CLISP transformations, so it does spelling correction, fixes 8-9 errors, handles F/L, etc. Thus you will see the same messages, and be asked for approval in the same situations, as you would if the expression were actually run. If DWIM is unable to make a correction, no message is printed, the form is left as it was, and the analysis proceeds.

DWIMIFY knows exactly how the interpreter works. It knows the syntax of PROGs, SELECTQs, LAMBDA expressions, SETQs, et al. It knows how variables are bound, and that the argument of NLAMBDAs are not evaluated (you can inform DWIMIFY of a function or macro’s nonstandard binding or evaluation by giving it a suitable INFO property, see below). In the course of its analysis of a particular expression, DWIMIFY builds a list of the bound variables from the LAMBDA expressions and PROGs that it encounters. It uses this list for spelling corrections. DWIMIFY also knows not to try to "correct" variables that are on this list since they would be bound if the expression were actually being run. However, note that DWIMIFY cannot, a priori, know about variables that are used freely but would be bound in a higher function if the expression were evaluated in its normal context. Therefore, DWIMIFY will try to "correct" these variables. Similarly, DWIMIFY will attempt to correct forms for which CAR is undefined, even when the form is not in error from your standpoint, but the corresponding function has simply not yet been defined.

Note: DWIMIFY rebinds FIXSPELLDEFAULT to N, so that if you are not at the terminal when DWIMIFYing (or compiling), spelling corrections will not be performed.

DWIMIFY will also inform you when it encounters an expression with too *many* arguments (unless DWIMCHECK#ARGSFLG = NIL), because such an occurrence, although does not cause an error in the Interlisp interpreter, nevertheless is frequently symptomatic of a parenthesis error. For example, if you wrote (CONS (QUOTE FOO X)) instead of (CONS (QUOTE FOO) X), DWIMIFY will print:

```
POSSIBLE PARENTHESIS ERROR IN
(QUOTE FOO X)
TOO MANY ARGUMENTS (MORE THAN 1)
```

DWIMIFY will also check to see if a PROG label contains a clisp character (unless DWIMCHECKPROGLABELSFLG = NIL, or the label is a member of NOFIXVARSLST), and if so, will alert you by printing the message SUSPICIOUS PROG LABEL, followed by the label. The PROG label will *not* be treated as CLISP.

Note that in most cases, an attempt to transform a form that is already as you intended will have no effect (because there will be nothing to which that form could reasonably be transformed). However, in order to avoid needless calls to DWIM or to avoid possible confusion, you can inform DWIMIFY *not* to attempt corrections or transformations on certain functions or variables by adding them to the list NOFIXFNSLST or NOFIXVARSLST respectively. Note that you could achieve the same effect by simply setting the corresponding variables, and giving the functions dummy definitions.

DWIMIFY will never attempt corrections on global variables, i.e., variables that are a member of the list GLOBALVARS, or have the property GLOBALVAR with value T, on their property list. Similarly, DWIMIFY will not attempt to correct variables declared to be SPECVARS in block declarations or via DECLARE expressions in the function body. You can also declare variables that are simply used freely in a function by using the USEDFREE declaration.

DWIMIFY and DWIMIFYFNS (used to DWIMIFY several functions) maintain two internal lists of those functions and variables for which corrections were unsuccessfully attempted. These lists are initialized to the values of NOFIXFNSLST and NOFIXVARSLST. Once an attempt is made to fix a particular function or variable, and the attempt fails, the function or variable is added to the corresponding list, so that on subsequent occurrences (within this call to DWIMIFY or DWIMIFYFNS), no attempt at correction is made. For example, if FOO calls FIE several times, and FIE is undefined at the time FOO is DWIMIFYed, DWIMIFY will not bother with FIE after the first occurrence. In other words, once DWIMIFY "notices" a function or variable, it no longer attempts to correct it. DWIMIFY and DWIMIFYFNS also "notice" free variables that are set in the expression being processed.

Moreover, once DWIMIFY "notices" such functions or variables, it subsequently treats them the same as though they were actually defined or set.

Note that these internal lists are local to each call to DWIMIFY and DWIMIFYFNS, so that if a function containing FOOO, a misspelled call to FOO, is DWIMIFYed before FOO is defined or mentioned, if the function is DWIMIFYed again after FOO has been defined, the correction will be made.

You can undo selected transformations performed by DWIMIFY, as described in Chapter 13.

(**DWIMIFY** *X QUIETFLG L*) [Function]

Performs all DWIM and CLISP corrections and transformations on *X* that would be performed if *X* were run, and prints the result unless *QUIETFLG* = T.

If *X* is an atom and *L* is NIL, *X* is treated as the name of a function, and its entire definition is DWIMIFYed. If *X* is a list or *L* is not NIL, *X* is the expression to be DWIMIFYed. If *L* is not NIL, it is the edit push-down list leading to *X*, and is used for determining context, i.e., what bound variables would be in effect when *X* was evaluated, whether *X* is a form or sequence of forms, e.g., a COND clause, etc.

If *X* is an iterative statement and *L* is NIL, DWIMIFY will also print the translation, i.e., what is stored in the hash array.

(**DWIMIFYFNS** FN_1 ... FN_N) [NLambda NoSpread Function]

DWIMIFYs each of the functions given. If only one argument is given, it is evalued. If its value is a list, the functions on this list are DWIMIFYed. If only one argument is given, it is atomic, its value is not a list, and it is the name of a known file, DWIMIFYFNS will operate on (FILEFNSLST FN_1), e.g. (DWIMIFYFNS FOO.LSP) will DWIMIFY every function in the file FOO.LSP.

Every 30 seconds, DWIMIFYFNS prints the name of the function it is processing, a la PRETTYPRINT.

Value is a list of the functions DWIMIFYed.

DWIMINMACROSFLG [Variable]

Controls how DWIMIFY treats the arguments in a "call" to a macro, i.e., where the CAR of the form is undefined, but has a macro definition. If DWIMINMACROSFLG is T, then macros are treated as LAMBDA functions, i.e., the arguments are assumed to be evaluated, which means that DWIMIFY will descend into the argument list. If DWIMINMACROSFLG is NIL, macros are treated as NLAMBDA functions. DWIMINMACROSFLG is initially T.

INFO [Property Name]

Used to inform DWIMIFY of nonstandard behavior of particular forms with respect to evaluation, binding of arguments, etc. The INFO property of a symbol is a single atom or list of atoms chosen from among the following:

EVAL Informs DWIMIFY (and CLISP and Masterscope) that an nlambda function *does* evaluate its arguments. Can also be placed on a macro name to override the behavior of DWIMINMACROSFLG = NIL.

NOEVAL Informs DWIMIFY that a macro does *not* evaluate all of its arguments, even when DWIMINMACROSFLG = T.

BINDS Placed on the INFO property of a function or the CAR of a special form to inform DWIMIFY that the function or form binds variables. In this case, DWIMIFY assumes that CADR of the form is the variable list, i.e., a list of symbols, or lists of the form (*VAL VALUE*). LAMBDA, NLAMBDA, PROG, and RESETVARS are handled in this fashion.

LABELS Informs CLISPIFY that the form interprets top-level symbols as labels, so that CLISPIFY will never introduce an atom (by packing) at the top level of the expression. PROG is handled in this fashion.

NOFIXFNSLST [Variable]

List of functions that DWIMIFY will not try to correct.

NOFIXVARSLST [Variable]

List of variables that DWIMIFY will not try to correct.

NOSPELLFLG [Variable]

If T, DWIMIFY will not perform any spelling corrections. Initially NIL. NOSPELLFLG is reset to T when compiling functions whose definitions are obtained from a file, as opposed to being in core.

CLISPHELPFLG [Variable]

If NIL, DWIMIFY will not ask you for approval of any CLISP transformations. Instead, in those situations where approval would be required, the effect is the same as though you had been asked and said NO. Initially T.

DWIMIFYCOMPFLG [Variable]

If T, DWIMIFY is called before compiling an expression. Initially NIL.

DWIMCHECK#ARGSFLG [Variable]

If T, causes DWIMIFY to check for too many arguments in a form. Initially T.

DWIMCHECKPROGLABELSFLG [Variable]

If T, causes DWIMIFY to check whether a PROG label contains a CLISP character. Initially T.

DWIMESSGAG [Variable]

If T, suppresses all DWIMIFY error messages. Initially NIL.

CLISPRETRANFLG [Variable]

If T, informs DWIMIFY to (re)translate all expressions which have remote translations in the CLISP hash array. Initially NIL.

CLISPIFY

CLISPIFY converts Interlisp expressions to CLISP. Note that the expression given to CLISPIFY need *not* have originally been input as CLISP, i.e., CLISPIFY can be used on functions that were written before CLISP was even implemented. CLISPIFY is cognizant of declaration rules as well as all of the precedence rules. For example, CLISPIFY will convert (IPLUS A (ITIMES B C)) into A+B*C, but (ITIMES A (IPLUS B C)) into A*(B+C). CLISPIFY handles such cases by first DWIMIFYing the expression. CLISPIFY also knows how to handle expressions consisting of a mixture of Interlisp and CLISP, e.g., (IPLUS A B*C) is converted to A+B*C, but (ITIMES A B+C) to (A*(B+C)). CLISPIFY converts calls to the six basic mapping functions, MAP, MAPC, MAPCAR, MAPLIST, MAPCONC, and MAPCON, into equivalent iterative statements. It also converts certain easily recognizable internal PROG loops to the corresponding iterative statements. CLISPIFY can convert all iterative statements input in CLISP back to CLISP, regardless of how complicated the translation was, because the original CLISP is saved.

CLISPIFY is not destructive to the original Interlisp expression, i.e., CLISPIFY produces a new expression without changing the original. The new expression may however contain some "pieces" of the original, since CLISPIFY attempts to minimize the number of CONSes by not copying structure whenever possible.

CLISPIFY will not convert expressions appearing as arguments to NLAMBDA functions, except for those functions whose INFO property is or contains the atom EVAL. CLISPIFY also contains built in information enabling it to process special forms such as PROG, SELECTQ, etc. If the INFO property is or contains the atom LABELS, CLISPIFY will never create an atom (by packing) at the top level of the expression. PROG is handled in this fashion.

Note: Disabling a CLISP operator with CLDISABLE (see the Miscellaneous Functions and Variables section) will also disable the corresponding CLISPIFY transformation. Thus, if ← is "turned off", A←B will not transform to (SETQ A B), nor vice versa.

(**CLISPIFY** *X EDITCHAIN*) [Function]

Clispifies *X*. If *X* is an atom and *EDITCHAIN* is NIL, *X* is treated as the name of a function, and its definition (or EXPR property) is clispified. After CLISPIFY has finished, *X* is redefined (using /PUTD) with its new CLISP definition. The value of CLISPIFY is *X*. If *X* is atomic and not the name of a function, spelling correction is attempted. If this fails, an error is generated.

If *X* is a list, or *EDITCHAIN* is not NIL, *X* itself is the expression to be clispified. If *EDITCHAIN* is not NIL, it is the edit push-down list leading to *X* and is used to determine context as with DWIMIFY, as well as to obtain the local declarations, if any. The value of CLISPIFY is the clispified version of *X*.

(**CLISPIFYFNS** FN_1 ... FN_N) [NLambda NoSpread Function]

Like DWIMIFYFNS except calls CLISPIFY instead of DWIMIFY.

CL:FLG [Variable]

Affects CLISPIFY's handling of forms beginning with CAR, CDR, ... CDDDDR, as well as pattern match and record expressions. If CL:FLG is NIL, these are not transformed into the equivalent : expressions. This will prevent CLISPIFY from constructing any expression employing a : infix operator, e.g., (CADR X) will not be transformed to X:2. If CL:FLG is T, CLISPIFY will convert to : notation only when the argument is atomic or a simple list (a function name and one atomic argument). If CL:FLG is ALL, CLISPIFY will convert to : expressions whenever possible.

CL:FLG is initially T.

CLREMPARSFLG [Variable]

If T, CLISPIFY will remove parentheses in certain cases from simple forms, where "simple" means a function name and one or two atomic arguments. For example, (COND ((ATOM X) --)) will CLISPIFY to (IF ATOM X THEN --). However, if CLREMPARSFLG is set to NIL, CLISPIFY will produce (IF (ATOM X) THEN --). Regardless of the flag setting, the expression can be input in either form.

CLREMPARSFLG is initially NIL.

CLISPIFYPACKFLG [Variable]

CLISPIFYPACKFLG affects the treatment of infix operators with atomic operands. If CLISPIFYPACKFLG is T, CLISPIFY will pack these into single atoms, e.g., (IPLUS A (ITIMES B C)) becomes A+B*C. If CLISPIFYPACKFLG is NIL, no packing is done, e.g., the above becomes A + B * C.

CLISPIFYPACKFLG is initially T.

CLISPIFYUSERFN [Variable]

If T, causes the function CLISPIFYUSERFN, which should be a function of one argument, to be called on each form (list) not otherwise recognized by CLISPIFY. If a non-NIL value is returned, it is treated as the clispified form. Initially NIL

Note that CLISPIFYUSERFN must be both set and defined to use this feature.

FUNNYATOMLST [Variable]

Suppose you have variables named A, B, and A*B. If CLISPIFY were to convert (ITIMES A B) to A*B, A*B would not translate back correctly to (ITIMES A B), since it would be the name of a variable, and therefore would not cause an error. You can prevent this from happening by adding A*B to the list FUNNYATOMLST. Then, (ITIMES A B) would CLISPIFY to A * B.

Note that A*B's appearance on FUNNYATOMLST would *not* enable DWIM and CLISP to decode A*B+C as (IPLUS A*B C); FUNNYATOMLST is used only by CLISPIFY. Thus, if an identifier contains a CLISP character, it should always be separated (with spaces) from other operators. For example, if X* is a variable, you should write (SETQ X* *FORM*) in CLISP as X* ←*FORM*, not X*←*FORM*. In general, it is best to avoid use of identifiers containing CLISP character operators as much as possible.

Miscellaneous Functions and Variables

CLISPFLG [Variable]

If CLISPFLG = NIL, disables all CLISP infix or prefix transformations (but does not affect IF/THEN/ELSE statements, or iterative statements).

If CLISPFLG = TYPE-IN, CLISP transformations are performed only on expressions that are typed in for evaluation, i.e., not on user programs.

If CLISPFLG = T, CLISP transformations are performed on all expressions.

The initial value for CLISPFLG is T. CLISPIFYing anything will cause CLISPFLG to be set to T.

CLISPCHARS [Variable]

A list of the operators that can appear in the interior of an atom. Currently (+ - * / ↑ ~ ' = ← : < > +- ~= @ !).

CLISPCHARRAY [Variable]

A bit table of the characters on CLISPCHARS used for calls to STRPOSL (Chapter 4). CLISPCHARRAY is initialized by performing (SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS)).

CLISPINFIXSPLST [Variable]

A list of infix operators used for spelling correction.

CLISPARRAY [Variable]

Hash array used for storing CLISP translations. CLISPARRAY is checked by FAULTEVAL and FAULTAPPLY on erroneous forms before calling DWIM, and by the compiler.

(**CLEARCLISPARRAY** *NAME* --) [Function]

Macro and CLISP expansions are cached in CLISPARRAY, the systems CLISP hash array. When anything changes that would invalidate an expansion, it needs to be removed from the cache. CLEARCLISPARRAY does this for you. The system does this automatically whenever you define redefine a CLISP or macro form. If you have changed something that a CLISP word or a macro depends on the system will not be able to detect this, so you will have to invalidate the cahce by calling CLEARCLISPARRAY. You can clear the whole cache by calling (CLRHASH CLISPARRAY).

(**CLISPTRAN** *X TRAN*) [Function]

Gives *X* the translation *TRAN* by storing (key *X*, value *TRAN*) in the hash array CLISPARRAY. CLISPTRAN is called for all CLISP translations, via a non-linked, external function call, so it can be advised.

(**CLISPDEC** *DECLST*) [Function]

Puts into effect the declarations in *DECLST*. CLISPDEC performs spelling corrections on words not recognized as declarations. CLISPDEC is undoable.

(**CLDISABLE** *OP*) [Function]

Disables the CLISP operator *OP*. For example, (CLDISABLE '-) makes - be just another character. CLDISABLE can be used on all CLISP operators, e.g., infix operators, prefix operators, iterative statement operators, etc. CLDISABLE is undoable.

> Note: Simply removing a character operator from CLISPCHARS will prevent it from being treated as a CLISP operator when it appears as part of an atom, but it will continue to be an operator when it appears as a separate atom, e.g. (FOO + X) vs FOO+X.

CLISPIFTRANFLG [Variable]

Affects handling of translations of IF-THEN-ELSE statements (see Chapter 9). If T, the translations are stored elsewhere, and the (modified) CLISP retained. If NIL, the corresponding COND expression replaces the CLISP. Initially T.

CLISPIFYPRETTYFLG [Variable]

If non-NIL, causes PRETTYPRINT (and therefore PP and MAKEFILE) to CLISPIFY selected function definitions before printing them according to the following interpretations of CLISPIFYPRETTYFLG:

ALL Clispify all functions.

T or **EXPRS** Clispify all functions currently defined as EXPRs.

CHANGES Clispify all functions marked as having been changed.

a list Clispify all functions in that list.

CLISPIFYPRETTYFLG is (temporarily) reset to T when MAKEFILE is called with the option CLISPIFY, and reset to CHANGES when the file being dumped has the property FILETYPE value CLISP. CLISPIFYPRETTYFLG is initially NIL.

Note: If CLISPIFYPRETTYFLG is non-NIL, and the only transformation performed by DWIM are well formed CLISP transformations, i.e., no spelling corrections, the function will *not* be marked as changed, since it would only have to be re-clispified and re-prettyprinted when the file was written out.

(**PPT** *X*) [NLambda NoSpread Function]

Both a function and an edit macro for prettyprinting translations. It performs a PP after first resetting PRETTYTRANFLG to T, thereby causing any translations to be printed instead of the corresponding CLISP.

CLISP: [Editor Command]

Edit macro that obtains the translation of the correct expression, if any, from CLISPARRAY, and calls EDITE on it.

CL [Editor Command]

Edit macro. Replaces current expression with CLISPIFYed current expression. Current expression can be an element or tail.

DW [Editor Command]

Edit macro. DWIMIFYs current expression, which can be an element (atom or list) or tail.

Both CL and DW can be called when the current expression is either an element or a tail and will work properly. Both consult the declarations in the function being edited, if any, and both are undoable.

(**LOWERCASE** *FLG*) [Function]

If *FLG* = T, LOWERCASE makes the necessary internal modifications so that CLISPIFY will use lower case versions of AND, OR, IF, THEN, ELSE, ELSEIF, and all i.s. operators. This produces more readable output. Note that you can always type in *either* upper or lower case (or a combination), regardless of the action of LOWERCASE. If *FLG* = NIL, CLISPIFY will use uppercase versions of AND, OR, et al. The value of LOWERCASE is its previous "setting". LOWERCASE is undoable. The initial setting for LOWERCASE is T.

CLISP Internal Conventions

CLISP is almost entirely table driven by the property lists of the corresponding infix or prefix operators. For example, much of the information used for translating the + infix operator is stored on the property list of the symbol "+". Thus it is relatively easy to add new infix or prefix operators or change old ones, simply by adding or changing selected property values. (There *is* some built in information for handling minus, :, ', and ~, i.e., you could not yourself add such "special" operators, although you can disable or redefine them.)

Global declarations operate by changing the LISPFN and CLISPINFIX properties of the appropriate operators.

CLISPTYPE [Property Name]

The property value of the property CLISPTYPE is the precedence number of the operator: higher values have higher precedence, i.e., are tighter. Note that the actual value is unimportant, only the value relative to other operators. For example, CLISPTYPE for :, ↑, and * are 14, 6, and 4 respectively. Operators with the same precedence group left to right, e.g., / also has precedence 4, so A/B*C is (A/B)*C.

An operator can have a different left and right precedence by making the value of CLISPTYPE be a dotted pair of two numbers, e.g., CLISPTYPE of ← is (8 . -12). In this case, CAR is the left precedence, and CDR the right, i.e., CAR is used when comparing with operators on the *left*, and CDR with operators on the *right*. For example, A*B←C+D is parsed as A*(B←(C+D)) because the left precedence of ← is 8, which is higher than that of *, which is 4. The right precedence of ← is -12, which is lower than that of +, which is 2.

If the CLISPTYPE property for any operator is removed, the corresponding CLISP transformation is disabled, as well as the inverse CLISPIFY transformation.

UNARYOP [Property Name]

The value of property UNARYOP must be T for unary operators or brackets. The operand is always on the right, i.e., unary operators or brackets are always prefix operators.

BROADSCOPE [Property Name]

The value of property BROADSCOPE is T if the operator has lower precedence than Interlisp forms, e.g., LT, EQUAL, AND, etc. For example, (FOO X AND Y) parses as ((FOO X) AND Y). If the BROADSCOPE property were removed from the property list of AND, (FOO X AND Y) would parse as (FOO (X AND Y)).

LISPFN [Property Name]

The value of the property LISPFN is the name of the function to which the infix operator translates. For example, the value of LISPFN for ↑ is EXPT, for ' QUOTE, etc. If the value of the property LISPFN is NIL, the infix operator itself is also the function, e.g., AND, OR, EQUAL.

SETFN [Property Name]

If FOO has a SETFN property FIE, then (FOO --)←X translates to (FIE -- X). For example, if you make ELT be an infix operator, e.g. #, by putting appropriate CLISPTYPE and LISPFN properties on the property list of # then you can also make # followed by ← translate to SETA, e.g., X#N←Y to (SETA X N Y), by putting SETA on the property list of

ELT under the property SETFN. Putting the list (ELT) on the property list of SETA under property SETFN will enable SETA forms to CLISPIFY back to ELT's.

CLISPINFIX [Property Name]

The value of this property is the CLISP infix to be used in CLISPIFYing. This property is stored on the property list of the corresponding Interlisp function, e.g., the value of property CLISPINFIX for EXPT is ↑, for QUOTE is ' etc.

CLISPWORD [Property Name]

Appears on the property list of clisp operators which can appear as CAR of a form, such as FETCH, REPLACE, IF, iterative statement operators, etc. Value of property is of the form (*KEYWORD* . *NAME*), where *NAME* is the lowercase version of the operator, and *KEYWORD* is its type, e.g. FORWORD, IFWORD, RECORDWORD, etc.

KEYWORD can also be the name of a function. When the atom appears as CAR of a form, the function is applied to the form and the result taken as the correct form. In this case, the function should either physically change the form, or call CLISPTRAN to store the translation.

As an example, to make & be an infix character operator meaning OR, you could do the following:

```
←(PUTPROP '& 'CLISPTYPE (GETPROP 'OR 'CLISPTYPE))
←(PUTPROP '& 'LISPFN 'OR)
←(PUTPROP '& 'BROADSCOPE T)
←(PUTPROP 'OR 'CLISPINFIX '&)
←(SETQ CLISPCHARS (CONS '& CLISPCHARS))
←(SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS))
```

21. PERFORMANCE ISSUES

This chapter describes a number of areas that often contribute to performance problems in Medley programs. Many performance problems can be improved by optimizing the use of storage, since allocating and reclaiming large amounts of storage is expensive. Another tactic that can sometimes yield performance improvements is to change the use of variable bindings on the stack to reduce variable lookup time. There are a number of tools that can be used to determine which parts of a computation cause performance bottlenecks.

Storage Allocation and Garbage Collection

As an Medley application program runs, it creates data structures (allocated out of free storage space), manipulates them, and then discards them. If there were no way to reclaim this space, over time the Medley memory would fill up, and the computation would come to a halt. Actually, long before this could happen the system would probably become intolerably slow, due to “data fragmentation,” which occurs when the data currently in use are spread over many virtual memory pages, so that most of the computer time must be spent swapping disk pages into physical memory. The problem of fragmentation will occur in any situation where the virtual memory is significantly larger than the real physical memory. To reduce swapping, you want to keep the "working set" (the set of pages containing actively referenced data) as small as possible.

You can write programs that don’t generate much “garbage” data, or which recycle data, but such programs tend to be complex and hard to debug. Spending effort writing such programs defeats the whole point of using a system with automatic storage allocation. An important part of any Lisp implementation is the “garbage collector” that finds discarded data and reclaims its space.

There are several well-known approaches to garbage collection. One method is the traditional mark-and-sweep, which identifies “garbage” data by marking all accessible data structures, and then sweeping through the data spaces to find all unmarked objects (i.e., not referenced by any other object). This method is guaranteed to reclaim all garbage, but it takes time proportional to the number of allocated objects, which may be very large. Also, the time that a mark-and-sweep garbage collection takes is independent of the amount of garbage collected; it is possible to sweep through the whole virtual memory, and only recover a small amount of garbage.

For interactive applications, it is not acceptable to have long interruptions in a computation for to garbage collect. Medley solves this problem by using a reference-counting garbage collector. With this scheme, there is a table containing counts of how many times each object is referenced. This table is updated as pointers are created and discarded, incurring a small overhead distributed over the computation as a whole. (Note: References from the stack are not counted, but are handled separately at "sweep" time; thus the vast majority of data manipulations do not cause updates to this table.) At opportune moments, the garbage collector scans this table, and reclaims all objects that are no longer accessible (have a reference count of zero). The pause while objects are reclaimed is only the time for scanning the reference count tables (small) plus time proportional to the amount of garbage that has to be collected (typically less than a second). “Opportune” times occur when a certain number of cells have been allocated or when the system has been waiting for you to type something for long enough. The frequency of garbage collection is controlled by the functions and variables described below. For the best system performance, it is desirable to adjust these parameters for frequent, short garbage collections, which will not interrupt interactive applications for very long, and which will have the added benefit of reducing data fragmentation, keeping the working set small.

One problem with the Medley garbage collector is that not all garbage is guaranteed to be collected. Circular data structures, which point to themselves directly or indirectly, are never reclaimed, since their reference counts are always at least one. With time, this unreclaimable garbage may increase the working set to unacceptable levels. Some users have worked with the same Medley virtual memory for a very long time, but it is a good idea to occasionally save all of your functions in files, reinitialize Medley, and rebuild your system. Many users end their working day by issuing a command to rebuild their system and then leaving the machine to perform this task in their absence. If the system seems to be spending too much time swapping (an indication of fragmented working set), this procedure is definitely recommended.

Another limitation of the reference-counting garbage collector is that the table in which reference counts are maintained is of fixed size. For typical Lisp objects that are pointed to from exactly one place (e.g., the individual conses in a list), no burden is placed on this table, since objects whose reference count is 1 are not explicitly represented in the table. However, large, "rich" data structures, with many interconnections, backward links, cross references, etc, can contribute many entries to the reference count table. For example, if you created a data structure that functioned as a doubly-linked list, such a structure would contribute an entry (reference count 2) for each element.

When the reference count table fills up, the garbage collector can no longer maintain consistent reference counts, so it stops doing so altogether. At this point, a window appears on the screen with the following message, and the debugger is entered:

```
Internal garbage collector tables have overflowed, due
to too many pointers with reference count greater than 1.
*** The garbage collector is now disabled. ***
Save your work and reload as soon as possible.
```

[This message is slightly misleading, in that it should say "count not equal to 1". In the current implementation, the garbage collection of a large pointer array whose elements are not otherwise pointed to can place a special burden on the table, as each element’s reference count simultaneously drops to zero and is thus added to the reference count table for the short period before the element is itself reclaimed.]

If you exit the debugger window (e.g., with the RETURN command), your computation can proceed; however, the garbage collector is no longer operating. Thus, your virtual memory will become cluttered with objects no longer accessible, and if you continue for long enough in the same virtual memory image you will eventually fill up the virtual memory backing store and grind to a halt.

Garbage collection in Medley is controlled by the following functions and variables:

(**RECLAIM**) [Function]

Initiates a garbage collection. Returns 0.

(**RECLAIMMIN** *N*) [Function]

Sets the frequency of garbage collection. Interlisp keeps track of the number of cells of any type that have been allocated; when it reaches a given number, a garbage collection occurs. If *N* is non-NIL, this number is set to *N*. Returns the current setting of the number.

RECLAIMWAIT [Variable]

Medley will invoke a RECLAIM if the system is idle and waiting for your input for RECLAIMWAIT seconds (currently set for 4 seconds).

(**GCGAG** *MESSAGE*) [Function]

Sets the behavior that occurs while a garbage collection is taking place. If *MESSAGE* is non-NIL, the cursor is complemented during a RECLAIM; if *MESSAGE* = NIL, nothing happens. The value of GCGAG is its previous setting.

(**GCTRP**) [Function]

Returns the number of cells until the next garbage collection, according to the RECLAIMMIN number.

The amount of storage allocated to different data types, how much of that storage is in use, and the amount of data fragmentation can be determined using the following function:

(**STORAGE** *TYPES PAGETHRESHOLD*) [Function]

STORAGE prints out a summary, for each data type, of the amount of space allocated to the data type, and how much of that space is currently in use. If *TYPES* is non-NIL, STORAGE only lists statistics for the specified types. *TYPES* can be a symbol or a list of types. If *PAGETHRESHOLD* is non-NIL, then STORAGE only lists statistics for types that have at least *PAGETHRESHOLD* pages allocated to them.

STORAGE prints out a table with the column headings Type, Assigned, Free Items, In use, and Total alloc. Type is the name of the data type. Assigned is how much of your virtual memory is set aside for items of this type. Currently, memory is allocated in quanta of two pages (1024 bytes). The numbers under Assigned show the number of pages and the total number of items that fit on those pages. Free Items shows how many items are available to be allocated (using the create construct, Chapter 8); these constitute the "free list" for that data type. In use shows how many items of this type are currently in use, i.e., have pointers to them and hence have not been garbage collected. If this number is higher than your program seems to warrant, you may want to look for storage leaks. The sum of Free Items and In use is always the same as the total Assigned items. Total alloc is the total number of items of this type that have ever been allocated (see BOXCOUNT, in the Performance Measuring section below).

> Note: The information about the number of items of type LISTP is only approximate, because list cells are allocated in a special way that precludes easy computation of the number of items per page.

> Note: When a data type is redeclared, the data type name is reassigned. Pages which were assigned to instances of the old data type are labeled **DEALLOC**.

At the end of the table printout, STORAGE prints a "Data Spaces Summary" listing the number of pages allocated to the major data areas in the virtual address space: the space for fixed-length items (including datatypes), the space for variable-length items, and the space for symbols. Variable-length data types such as arrays have fixed-length "headers," which is why they also appear in the printout of fixed-length data types. Thus, the line printed for the BITMAP data type says how many bitmaps have been allocated, but the "assigned pages" column counts only the headers, not the space used by the variable-length part of the bitmap. This summary also lists "Remaining Pages" in relation to the largest possible virtual memory, not the size of the virtual memory backing file in use. This file may fill up, causing a STORAGE FULL error, long before the "Remaining Pages" numbers reach zero.

STORAGE also prints out information about the sizes of the entries on the variable-length data free list. The block sizes are broken down by the value of the variable STORAGE.ARRAYSIZES, initially (4 16 64 256 1024 4096 16384 NIL), which yields a printout of the form:

```
variable-datum free list:
le 4        26 items;    104 cells.
le 16       72 items;    783 cells.
le 64       36 items;    964 cells.
le 256      28 items;   3155 cells.
le 1024      3 items;   1175 cells.
le 4096      5 items;   8303 cells.
le 16384     3 items;  17067 cells.
others       1 items;  17559 cells.
```

This information can be useful in determining if the variable-length data space is fragmented. If most of the free space is composed of small items, then the allocator may not be able to find room for large items, and will extend the variable datum space. If this is extended too much, this could cause an ARRAYS FULL error, even if there is a lot of space left in little chunks.

(**STORAGE.LEFT**) [Function]

Provides a programmatic way of determining how much storage is left in the major data areas in the virtual address space. Returns a list of the form (*MDSFREE MDSFRAC 8MBFRAC ATOMFREE ATOMFRAC*), where the elements are interpreted as follows:

MDSFREE The number of free pages left in the main data space (which includes both fixed-length and variable-length data types).

MDSFRAC The fraction of the total possible main data space that is free.

8MBFRAC The fraction of the total main data space that is free, relative to eight megabytes.

This number is useful when using Medley on some early computers where the hardware limits the address space to eight megabytes. The function 32MBADDRESSABLE returns non-NIL if the currently running Medley system can use the full 32 megabyte address space.

ATOMFREE The number of free pages left in the symbol space.

ATOMFRAC The fraction of the total symbol space that is free.

Note: Another important space resource is the amount of the virtual memory backing file in use (see VMEMSIZE, Chapter 12). The system will crash if the virtual memory file is full, even if the address space is not exhausted.

Variable Bindings

Different implementations of Lisp use different methods of accessing free variables. The binding of variables occurs when a function or a PROG is entered. For example, if the function FOO has the definition (LAMBDA (A B) *BODY*), the variables A and B are bound so that any reference to A or B from *BODY* or any function called from *BODY* will refer to the arguments to the function FOO and not to the value of A or B from a higher level function. All variable names (symbols) have a top level value cell which is used if the variable has not been bound in any function. In discussions of variable access, it is useful to distinguish between three types of variable access: local, special and global. Local variable access is the use of a variable that is bound within the function from which it is used. Special variable access is the use of a variable that is bound by another function. Global variable access is the use of a variable that has not been bound in any function. We will often refer to a variable all of whose accesses are local as a "local variable." Similarly, a variable all of whose accesses are global we call a "global variable."

In a “deep” bound system, a variable is bound by saving on the stack the variable’s name together with a value cell which contains that variable’s new value. When a variable is accessed, its value is found by searching the stack for the most recent binding (occurrence) and retrieving the value stored there. If the variable is not found on the stack, the variable’s top level value cell is used.

In a “shallow” bound system, a variable is bound by saving on the stack the variable name and the variable’s old value and putting the new value in the variable’s top level value cell. When a variable is accessed, its value is always found in its top level value cell.

The deep binding scheme has one disadvantage: the amount of cpu time required to fetch the value of a variable depends on the stack distance between its use and its binding. The compiler can determine local variable accesses and compiles them as fetches directly from the stack. Thus this computation cost only arises in the use of variable not bound in the local frame ("free" variables). The process of finding the value of a free variable is called free variable lookup.

In a shallow bound system, the amount of cpu time required to fetch the value of a variable is constant regardless of whether the variable is local, special or global. The disadvantages of this scheme are that the actual binding of a variable takes longer (thus slowing down function call), the cells that contain the current in use values are spread throughout the space of all symbol value cells (thus increasing the working set size of functions) and context switching between processes requires unwinding and rewinding the stack (thus effectively prohibiting the use of context switching for many applications).

Medley uses deep binding, because of the working set considerations and the speed of context switching. The free variable lookup routine is microcoded, thus greatly reducing the search time. In benchmarks, the largest percentage of free variable lookup time was 20 percent of the total ellapsed time; the normal time was between 5 and 10 percent.

Because of the deep binding, you can sometimes significantly improve performance by declaring global variables. If a variable is declared global, the compiler will compile an access to that variable as a retrieval of its top level value, completely bypassing a stack search. This should be done only for variables that are never bound in functions, such as global databases and flags.

Global variable declarations should be done using the GLOBALVARS file manager command (Chapter 17). Its form is (GLOBALVARS *VAR*$_1$ ... *VAR*$_N$).

Another way of improving performance is to declare variables as local within a function. Normally, all variables bound within a function have their names put on the stack, and these names are scanned during free variable lookup. If a variable is declared to be local within a function, its name is not put on the stack, so it is not scanned during free variable lookup, which may increase the speed of lookups. The compiler can also make some other optimizations if a variable is known to be local to a function.

A variable may be declared as local within a function by including the form (DECLARE (LOCALVARS *VAR*$_1$ ... *VAR*$_N$)) following the argument list in the definition of the function. Local variable declarations only effect the compilation of a function. Interpreted functions put all of their variable names on the stack, regardless of any declarations.

Performance Measuring

This section describes functions that gather and display statistics about a computation, such as as the elapsed time, and the number of data objects of different types allocated. TIMEALL and TIME gather statistics on the evaluation of a specified form. BREAKDOWN gathers statistics on individual functions called during a computation. These functions can be used to determine which parts of a computation are consuming the most resources (time, storage, etc.), and could most profitably be improved.

(**TIMEALL** *TIMEFORM NUMBEROFTIMES TIMEWHAT INTERPFLG*) [NLambda Function]

Evaluates the form *TIMEFORM* and prints statistics on time spent in various categories (elapsed, keyboard wait, swapping time, gc) and data type allocation.

For more accurate measurement on small computations, *NUMBEROFTIMES* may be specified (its default is 1) to cause *TIMEFORM* to be executed *NUMBEROFTIMES* times. To improve the accuracy of timing open-coded operations in this case, TIMEALL compiles a form to execute *TIMEFORM NUMBEROFTIMES* times (unless *INTERPFLG* is non-NIL), and then times the execution of the compiled form.

> Note: If TIMEALL is called with *NUMBEROFTIMES* > 1, the dummy form is compiled with compiler optimizations on. This means that it is not meaningful to use TIMEALL with very simple forms that are optimized out by the compiler. For example, `(TIMEALL '(IPLUS 2 3) 1000)` will time a compiled function which simply returns the number 5, since `(IPLUS 2 3)` is optimized to the integer 5.

TIMEWHAT restricts the statistics to specific categories. It can be an atom or list of datatypes to monitor, and/or the atom TIME to monitor time spent. Note that ordinarily, TIMEALL monitors all time and datatype usage, so this argument is rarely needed.

TIMEALL returns the value of the last evaluation of *TIMEFORM*.

(**TIME** *TIMEX TIMEN TIMETYP*) [NLambda Function]

TIME evaluates the form *TIMEX*, and prints out the number of CONS cells allocated and computation time. Garbage collection time is subtracted out. This function has been largely replaced by TIMEALL.

If *TIMEN* is greater than 1, *TIMEX* is executed *TIMEN* times, and TIME prints out (number of conses)/*TIMEN*, and (computation time)/*TIMEN*. If *TIMEN* = NIL, it defaults to 1. This is useful for more accurate measurement on small computations.

If *TIMETYP* is 0, TIME measures and prints total *real* time as well as computation time. If *TIMETYP* = 3, TIME measures and prints garbage collection time as well as computation time. If *TIMETYP* = T, TIME measures and prints the number of pagefaults.

TIME returns the value of the last evaluation of *TIMEX*.

(**BOXCOUNT** *TYPE N*) [Function]

Returns the number of data objects of type *TYPE* allocated since this Interlisp system was created. *TYPE* can be any data type name (see TYPENAME, Chapter 8). If *TYPE* is NIL, it defaults to FIXP. If *N* is non-NIL, the corresponding counter is reset to *N*.

(**CONSCOUNT** *N*) [Function]

Returns the number of CONS cells allocated since this Interlisp system was created. If *N* is non-NIL, resets the counter to *N*. Equivalent to `(BOXCOUNT 'LISTP N)`.

(**PAGEFAULTS**) [Function]

Returns the number of page faults since this Interlisp system was created.

BREAKDOWN

TIMEALL collects statistics for whole computations. BREAKDOWN is available to analyze the breakdown of computation time (or any other measureable quantity) function by function.

(**BREAKDOWN** *FN₁ . . . FN_N*) [NLambda NoSpread Function]

You call BREAKDOWN giving it a list of function names (unevaluated). These functions are modified so that they keep track of various statistics.

To remove functions from those being monitored, simply UNBREAK (Chapter 15) the functions, thereby restoring them to their original state. To add functions, call BREAKDOWN on the new functions. This will not reset the counters for any functions not on the new list. However (BREAKDOWN) will zero the counters of all functions being monitored.

The procedure used for measuring is such that if one function calls other and both are "broken down", then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.

BREAKDOWN will *not* give accurate results if a function being measured is not returned from normally, e.g., a lower RETFROM (or ERROR) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

(**BRKDWNRESULTS** *RETURNVALUESFLG*) [Function]

BRKDWNRESULTS prints the analysis of the statistics requested as well as the number of calls to each function. If *RETURNVALUESFLG* is non-NIL, BRKDWNRESULTS will not to print the results, but instead return them in the form of a list of elements of the form (*FNNAME #CALLS VALUE*).

Example:

```
←  (BREAKDOWN SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←(PRETTYDEF '(SUPERPRINT) 'FOO)
FOO.;3
←(BRKDWNRESULTS)
FUNCTIONS   TIME    #CALLS  PER CALL   %
SUPERPRINT  8.261    365    0.023     20
SUBPRINT   31.910    141    0.226     76
COMMENT1    1.612      8    0.201      4
TOTAL      41.783    514    0.081
NIL
←(BRKDWNRESULTS T)
```

```
((SUPERPRINT 365 8261) (SUBPRINT 141 31910)
(COMMENT1 8 1612))
```

BREAKDOWN can be used to measure other statistics, by setting the following variables:

BRKDWNTYPE [Variable]

To use BREAKDOWN to measure other statistics, before calling BREAKDOWN, set the variable BRKDWNTYPE to the quantity of interest, e.g., TIME, CONSES, etc, or a list of such quantities. Whenever BREAKDOWN is called with BRKDWNTYPE not NIL, BREAKDOWN performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with a particular statistic, a measuring function will be defined, and the compiler will be called to compile it. The functions being broken down will be redefined to call this measuring function. When BREAKDOWN is through initializing, it sets BRKDWNTYPE back to NIL. Subsequent calls to BREAKDOWN will measure the new statistic until BRKDWNTYPE is again set and a new BREAKDOWN performed.

BRKDWNTYPES [Variable]

The list BRKDWNTYPES contains the information used to analyze new statistics. Each entry on BRKDWNTYPES should be of the form (*TYPE FORM FUNCTION*), where *TYPE* is a statistic name (as would appear in BRKDWNTYPE), *FORM* computes the statistic, and *FUNCTION* (optional) converts the value of form to some more interesting quantity. For example, (TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000))) measures computation time and reports the result in seconds instead of milliseconds. BRKDWNTYPES currently contains entries for TIME, CONSES, PAGEFAULTS, BOXES, and FBOXES.

Example:

```
←(SETQ BRKDWNTYPE '(TIME CONSES))
(TIME CONSES)
←(BREAKDOWN MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←(FLIP '(A B C D E F G H C Z) '(.. $1 .. #2 ..)
'(.. #3 ..))
(A B D E F G H Z)
←(BRKDWNRESULTS)
FUNCTIONS  TIME    #CALLS  PER CALL   %
MATCH      0.036    1       0.036    54
CONSTRUCT  0.031    1       0.031    46
TOTAL      0.067    2       0.033
FUNCTIONS  CONSES  #CALLS  PER CALL   %
MATCH      32       1      32.000    40
CONSTRUCT  49       1      49.000    60
TOTAL      81       2      40.500
NIL
```

Occasionally, a function being analyzed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If you were using TIME, you would specify a value for *TIMEN* greater than 1 to give greater accuracy. A similar option is available for BREAKDOWN. You can specify that a function(s) be executed a multiple number of times for each measurement, and the average value reported, by including a number in the list of functions given to BREAKDOWN. For example, BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP) means normal breakdown for EDITCOM and EDIT4F but executes (the body of) EDIT4E and EQP 10 times each time they are called. Of course, the functions so measured must not cause any harmful side effects, since they are executed more than once for each call. The printout from BRKDWNRESULTS will look the same as though each function were run only once, except that the measurement will be more accurate.

Another way of obtaining more accurate measurement is to expand the call to the measuring function in-line. If the value of BRKDWNCOMPFLG is non-NIL (initially NIL), then whenever a function is broken-down, it will be redefined to call the measuring function, and then recompiled. The measuring function is expanded in-line via an appropriate macro. In addition, whenever BRKDWNTYPE is reset, the compiler is called for *all* functions for which BRKDWNCOMPFLG was set at the time they were originally broken-down, i.e. the setting of the flag at the time a function is broken-down determines whether the call to the measuring code is compiled in-line.

GAINSPACE

If you have large programs and databases, you may sometimes find yourself in a situation where you need to obtain more space, and are willing to pay the price of eliminating some or all of the context information that the various user-assistance facilities such as the programmer's assistant, file package, CLISP, etc., have accumulated during the course of his session. The function GAINSPACE provides an easy way to selectively throw away accumulated data:

(**GAINSPACE**) [Function]

Prints a list of deletable objects, allowing you to specify at each point what should be discarded and what should be retained. For example:

```
←(GAINSPACE)
purge history lists ? Yes
purge everything, or just the properties, e.g.,
SIDE, LISPXPRINT, etc. ?
just the properties
discard definitions on property lists ? Yes
discard old values of variables ? Yes
erase properties ? No
erase CLISP translations? Yes
```

GAINSPACE is driven by the list GAINSPACEFORMS. Each element on GAINSPACEFORMS is of the form (*PRECHECK MESSAGE FORM KEYLST*). If *PRECHECK*, when evaluated, returns NIL, GAINSPACE skips to the next entry. For example, you will not be asked whether or not to purge the history list if it is not enabled. Otherwise, ASKUSER (Chapter 26) is called with the indicated *MESSAGE* and the (optional) *KEYLST*. If you respond No, i.e., ASKUSER returns N, GAINSPACE skips to the next entry. Otherwise, *FORM* is evaluated with the variable RESPONSE bound to the value of ASKUSER. In the above example, the *FORM* for the "purge history lists" question calls ASKUSER to ask "purge everything, ..." only if you had responded Yes. If you had responded with Everything, the second question would not have been asked.

The "erase properties" question is driven by a list SMASHPROPSMENU. Each element on this list is of the form (*MESSAGE* . *PROPS*). You are prompted with *MESSAGE* (by ASKUSER), and if your response is Yes, *PROPS* is added to the list SMASHPROPS. The "discard definitions on property lists" and "discard old values of variables" questions also add to SMASHPROPS. You will not be prompted for any entry on SMASHPROPSMENU for which all of the corresponding properties are already on SMASHPROPS. SMASHPROPS is initially set to the value of SMASHPROPSLST. This permits you to specify in advance those properties which you always want discarded, and not be asked about them subsequently. After finishing all the entries on GAINSPACEFORMS, GAINSPACE checks to see if the value of SMASHPROPS is non-NIL, and if so, does a MAPATOMS, i.e., looks at every atom in the system, and erases the indicated properties.

You can change or add new entries to GAINSPACEFORMS or SMASHPROPSMENU, so that GAINSPACE can also be used to purge structures that your programs have accumulated.

Using Data Types Instead of Records

If a program uses large numbers of large data structures, there are several advantages to representing them as user data types rather than as list structures. The primary advantage is increased speed: accessing and setting the fields of a data type can be significantly faster than walking through a list with repeated CARs and CDRs. Also,

Compiled code for referencing data types is usually smaller. Finally, by reducing the number of objects created (one object against many list cells), this can reduce the expense of garbage collection.

User data types are declared by using the DATATYPE record type (Chapter 8). If a list structure has been defined using the RECORD record type (Chapter 8), and all accessing operations are written using the record package's fetch, replace, and create operations, changing from RECORDs to DATATYPEs only requires editing the record declaration (using EDITREC, Chapter 8) to replace declaration type RECORD by DATATYPE, and recompiling.

> Note: There are some minor disadvantages: First, there is an upper limit on the number of data types that can exist. Also, space for data types is allocated two pages at a time. Each data type which has any instances allocated has at least two pages assigned to it, which may be wasteful of space if there are only a few examples of a given data type. These problems should not effect most applications programs.

Using “Fast” and “Destructive” Functions

Among the functions used for manipulating objects of various data types, there are a number of functions which have "fast" and "destructive" versions. You should be aware of what these functions do, and when they should be used.

“Fast” functions: By convention, a function named by prefixing an existing function name with F indicates that the new function is a "fast" version of the old. These usually have the same definitions as the slower versions, but they compile open and run without any "safety" error checks. For example, FNTH runs faster than NTH, however, it does not make as many checks (for lists ending with anything but NIL, etc). If these functions are given arguments that are not in the form that they expect, their behavior is unpredictable; they may run forever, or cause a system error. In general, you should only use "fast" functions in code that has already been completely debugged, to speed it up.

“Destructive” functions: By convention, a function named by prefixing an existing function with D indicates the new function is a "destructive" version of the old one, which does not make any new structure but cannibalizes its argument(s). For example, REMOVE returns a copy of a list with a particular element removed, but DREMOVE actually changes the list structure of the list. (Unfortunately, not all destructive functions follow this naming convention: the destructive version of APPEND is NCONC.) You should be careful when using destructive functions that they do not inadvertantly change data structures.

22. PERFORMANCE ISSUES

This chapter describes a number of areas that often contribute to performance problems in Interlisp-D programs. Many performance problems can be improved by optimizing the use of storage, since allocating and reclaiming large amounts of storage is expensive. Another tactic that can sometimes yield performance improvements is to change the use of variable bindings on the stack to reduce variable lookup time. There are a number of tools that can be used to determine which parts of a computation cause performance bottlenecks.

Storage Allocation and Garbage Collection

As an Interlisp-D applications program runs, it creates data structures (allocated out of free storage space), manipulates them, and then discards them. If there were no way of reclaiming this space, over time the Interlisp-D memory (both the physical memory in the machine and the virtual memory stored on the disk) would fill up, and the computation would come to a halt. Actually, long before this could happen the system would probably become intolerably slow, due to "data fragmentation," which occurs when the data currently in use are spread over many virtual memory pages, so that most of the computer time must be spent swapping disk pages into physical memory. The problem of fragmentation will occur in any situation where the virtual memory is significantly larger than the real physical memory. To reduce swapping, it is desirable to keep the "working set" (the set of pages containing actively referenced data) as small as possible.

It is possible to write programs that don't generate much "garbage" data, or which recycle data, but such programs tend to be overly complicated and difficult to debug. Spending effort writing such programs defeats the whole point of using a system with automatic storage allocation. An important part of any Lisp implementation is the "garbage collector" which identifies discarded data and reclaims its space. There are several well-known approaches to garbage collection. One method is the traditional mark-and-sweep garbage collection algorithm, which identifies "garbage" data by marking all accessible data structures, and then sweeping through the data spaces to find all unmarked objects (i.e., not referenced by any other object). Although this method is guaranteed to reclaim all garbage, it takes time proportional to the number of allocated objects, which may be very large. (Some allocated objects will have been marked during the "mark" phase, and the remainder will be collected during the "sweep" phase; so all will have to be touched in some way.) Also, the time that a mark-and-sweep garbage collection takes is independent of the amount of garbage collected; it is possible to sweep through the whole virtual memory, and only recover a small amount of garbage.

For interactive applications, it is not acceptable to have long interruptions in a computation for the purpose of garbage collection. Interlisp-D solves this problem by using a reference-counting garbage collector. With this scheme, there is a table containing counts of how many times each object is referenced. This table is incrementally updated as pointers are created and discarded, incurring a small overhead distributed over the computation as a whole. (Note: References from the stack are not counted, but are handled separately at "sweep" time; thus the vast majority of data manipulations do not cause updates to this table.) At opportune moments, the garbage collector scans this table, and reclaims all objects that are no longer accessible (have a reference count of zero). The pause while objects are reclaimed is only the time for scanning the reference count tables (small) plus time proportional to the amount of garbage that has to be collected (typically less than a second). "Opportune" times occur when a certain number of cells have been allocated or when the system has been waiting for the user to type something for long enough. The frequency of garbage collection is controlled by the functions and variables described below. For the best system performance, it is desirable to adjust these parameters for frequent, short garbage collections, which will not interrupt interactive applications for very long, and which will have the added benefit of reducing data fragmentation, keeping the working set small.

One problem with the Interlisp-D garbage collector is that not all garbage is guaranteed to be collected. Circular data structures, which point to themselves directly or indirectly, are never reclaimed, since their reference counts are always at least one. With time, this unreclaimable garbage may increase the working set to unacceptable levels. Some users have worked with the same Interlisp-D virtual memory for a very long time, but it is a good idea to occasionally save all of your functions in files, reinitialize Interlisp-D, and rebuild your system. Many users end their working day by issuing a command to rebuild their system and then leaving the machine to perform this task in their absence. If the system seems to be spending too much time swapping (an indication of fragmented working set), this procedure is definitely recommended.

Garbage collection in Interlisp-D is controlled by the following functions and variables:

`(RECLAIM)` [Function]

Initiates a garbage collection. Returns 0.

`(RECLAIMMIN` *N*`)` [Function]

Sets the frequency of garbage collection. Interlisp keeps track of the number of cells of any type that have been allocated; when it reaches a given number, a garbage collection occurs. If *N* is non-`NIL`, this number is set to *N*. Returns the current setting of the number.

RECLAIMWAIT [Variable]

Interlisp-D will invoke a RECLAIM if the system is idle and waiting for your input for RECLAIMWAIT seconds (currently set for 4 seconds).

(GCGAG *MESSAGE*) [Function]

Sets the behavior that occurs while a garbage collection is taking place. If *MESSAGE* is non-NIL, the cursor is complemented during a RECLAIM; if *MESSAGE*=NIL, nothing happens. The value of GCGAG is its previous setting.

(GCTRP) [Function]

Returns the number of cells until the next garbage collection, according to the RECLAIMMIN number.

The amount of storage allocated to different data types, how much of that storage is in use, and the amount of data fragmentation can be determined using the following function:

(STORAGE *TYPES PAGETHRESHOLD*) [Function]

STORAGE prints out a summary, for each data type, of the amount of space allocated to the data type, and how much of that space is currently in use. If *TYPES* is non-NIL, STORAGE only lists statistics for the specified types. *TYPES* can be a litatom or a list of types. If *PAGETHRESHOLD* is non-NIL, then STORAGE only lists statistics for types that have at least *PAGETHRESHOLD* pages allocated to them.

STORAGE prints out a table with the column headings **Type**, **Assigned**, **Free Items**, **In use**, and **Total alloc**. **Type** is the name of the data type. **Assigned** is how much of your virtual memory is set aside for items of this type. Currently, memory is allocated in quanta of two pages (1024 bytes). The numbers under **Assigned** show the number of pages and the total number of items that fit on those pages. **Free Items** shows how many items are available to be allocated (using the **create** construct, Chapter 8); these constitute the "free list" for that data type. **In use** shows how many items of this type are currently in use, i.e., have pointers to them and hence have not been garbage collected. If this number is higher than your program seems to warrant, you may want to look for storage leaks. The sum of **Free Items** and **In use** is always the same as the total **Assigned** items. **Total alloc** is the total number of items of this type that have ever been allocated (see BOXCOUNT, in the Performance Measuring section below).

Note: The information about the number of items of type LISTP is only approximate, because list cells are allocated in a special way that precludes easy computation of the number of items per page.

Note: When a data type is redeclared, the data type name is reassigned. Pages which were assigned to instances of the old data type are labeled ****DEALLOC****.

At the end of the table printout, STORAGE prints a "Data Spaces Summary" listing the number of pages allocated to the major data areas in the virtual address space: the space for fixed-length items (including datatypes), the space for variable-length items, and the space for litatoms. Variable-length data types such as arrays have fixed-length "headers," which is why they also appear in the printout of fixed-length data types. Thus, the line printed for the BITMAP data type says how many bitmaps have been allocated, but the "assigned pages" column counts only the headers, not the space used by the variable-length part of the bitmap. This summary also lists "Remaining Pages" in relation to the largest possible virtual memory, not the size of the virtual memory backing file in use. This file may fill up, causing a **STORAGE FULL** error, long before the "Remaining Pages" numbers reach zero.

STORAGE also prints out information about the sizes of the entries on the variable-length data free list. The block sizes are broken down by the value of the variable STORAGE.ARRAYSIZES, initially (4 16 64 256 1024 4096 16384 NIL), which yields a printout of the form:

```
variable-datum free list:
le 4        26 items;    104 cells.
le 16       72 items;    783 cells.
le 64       36 items;    964 cells.
le 256      28 items;   3155 cells.
le 1024      3 items;   1175 cells.
```

```
le 4096      5 items;   8303 cells.
le 16384     3 items;  17067 cells.
others       1 items;  17559 cells.
```

This information can be useful in determining if the variable-length data space is fragmented. If most of the free space is composed of small items, then the allocator may not be able to find room for large items, and will extend the variable datum space. If this is extended too much, this could cause an ARRAYS FULL error, even if there is a lot of space left in little chunks.

`(STORAGE.LEFT)` [Function]

Provides a programmatic way of determining how much storage is left in the major data areas in the virtual address space. Returns a list of the form (*MDSFREE MDSFRAC 8MBFRAC ATOMFREE ATOMFRAC*), where the elements are interpreted as follows:

MDSFREE The number of free pages left in the main data space (which includes both fixed-length and variable-length data types).

MDSFRAC The fraction of the total possible main data space that is free.

8MBFRAC The fraction of the total main data space that is free, relative to eight megabytes.

This number is useful when using Interlisp-D on some early computers where the hardware limits the address space to eight megabytes. The function `32MBADDRESSABLE` returns non-`NIL` if the currently running Interlisp-D system can use the full 32 megabyte address space.

ATOMFREE The number of free pages left in the litatom space.

ATOMFRAC The fraction of the total litatom space that is free.

Note: Another important space resource is the amount of the virtual memory backing file in use (see `VMEMSIZE`, Chapter 12). The system will crash if the virtual memory file is full, even if the address space is not exhausted.

Variable Bindings

Different implementations of lisp use different methods of accessing free variables. The binding of variables occurs when a function or a `PROG` is entered. For example, if the function `FOO` has the definition `(LAMBDA (A B)` *BODY*`)`, the variables `A` and `B` are bound so that any reference to `A` or `B` from *BODY* or any function called from *BODY* will refer to the arguments to the function `FOO` and not to the value of `A` or `B` from a higher level function. All variable names (litatoms) have a top level value cell which is used if the variable has not been bound in any function. In discussions of variable access, it is useful to distinquish between three types of variable access: local, special and global. Local variable access is the use of a variable that is bound within the function from which it is used. Special variable access is the use of a variable that is bound by another function. Global variable access is the use of a variable that has not been bound in any function. We will often refer to a variable all of whose accesses are local as a "local variable." Similarly, a variable all of whose accesses are global we call a "global variable."

In a "deep" bound system, a variable is bound by saving on the stack the variable's name together with a value cell which contains that variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding (occurrence) and retrieving the value stored there. If the variable is not found on the stack, the variable's top level value cell is used.

In a "shallow" bound system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's top level value cell. When a variable is accessed, its value is always found in its top level value cell.

The deep binding scheme has one disadvantage: the amount of cpu time required to fetch the value of a variable depends on the stack distance between its use and its binding. The compiler can determine local variable accesses and compiles them as fetches directly from the stack. Thus this computation cost only arises in the use of variable not bound in the local frame ("free" variables). The process of finding the value of a free variable is called free variable lookup.

In a shallow bound system, the amount of cpu time required to fetch the value of a variable is constant regardless of whether the variable is local, special or global. The disadvantages of this scheme are that the actual binding of a variable takes longer (thus slowing down function call), the cells that contain the current in use values are spread throughout the space of all litatom value cells (thus increasing the working set size of functions) and context switching between processes requires unwinding and rewinding the stack (thus effectively prohibiting the use of context switching for many applications).

Interlisp-D uses deep binding, because of the working set considerations and the speed of context switching. The free variable lookup routine is microcoded, thus greatly reducing the search time. In benchmarks, the largest percentage of free variable lookup time was 20 percent of the total ellapsed time; the normal time was between 5 and 10 percent.

One consequence of Interlisp-D's deep binding scheme is that users may significantly improve performance by declaring global variables in certain situations. If a variable is declared global, the compiler will compile an access to that variable as a retrieval of its top level value, completely bypassing a stack search. This should be done only for variables that are never bound in functions, such as global databases and flags.

Global variable declarations should be done using the `GLOBALVARS` file package command (Chapter 17). Its form is `(GLOBALVARS` VAR_1 ... VAR_N`)`.

Another way of improving performance is to declare variables as local within a function. Normally, all variables bound within a function have their names put on the stack, and these names are scanned during free variable lookup. If a variable is declared to be local within a function, its name is not put on the stack, so it is not scanned during free variable lookup, which may increase the speed of lookups. The compiler can also make some other optimizations if a variable is known to be local to a function.

A variable may be declared as local within a function by including the form `(DECLARE (LOCALVARS` VAR_1 ... VAR_N`))` following the argument list in the definition of the function. Local variable declarations only effect the compilation of a function. Interpreted functions put all of their variable names on the stack, regardless of any declarations.

Performance Measuring

This section describes functions that gather and display statistics about a computation, such as as the elapsed time, and the number of data objects of different types allocated. `TIMEALL` and `TIME` gather statistics on the evaluation of a specified form. `BREAKDOWN` gathers statistics on individual functions called during a computation. These functions can be used to determine which parts of a computation are consuming the most resources (time, storage, etc.), and could most profitably be improved.

(`TIMEALL` *TIMEFORM NUMBEROFTIMES TIMEWHAT INTERPFLG* —) [NLambda Function]

Evaluates the form *TIMEFORM* and prints statistics on time spent in various categories (elapsed, keyboard wait, swapping time, gc) and data type allocation.

For more accurate measurement on small computations, *NUMBEROFTIMES* may be specified (its default is 1) to cause *TIMEFORM* to be executed *NUMBEROFTIMES* times. To improve the accuracy of timing open-coded operations in this case, `TIMEALL` compiles a form to execute *TIMEFORM* *NUMBEROFTIMES* times (unless *INTERPFLG* is non-`NIL`), and then times the execution of the compiled form.

Note: If `TIMEALL` is called with *NUMBEROFTIMES*>1, the dummy form is compiled with compiler optimizations on. This means that it is not meaningful to use `TIMEALL` with very simple forms that are optimized out by the compiler. For example, `(TIMEALL '(IPLUS 2 3) 1000)` will time a compiled function which simply returns the number 5, since `(IPLUS 2 3)` is optimized to the integer 5.

TIMEWHAT restricts the statistics to specific categories. It can be an atom or list of datatypes to monitor, and/or the atom `TIME` to monitor time spent. Note that ordinarily, `TIMEALL` monitors all time and datatype usage, so this argument is rarely needed.

`TIMEALL` returns the value of the last evaluation of *TIMEFORM*.

(`TIME` *TIMEX TIMEN TIMETYP*) [NLambda Function]

`TIME` evaluates the form *TIMEX*, and prints out the number of `CONS` cells allocated and computation time. Garbage collection time is subtracted out. This function has been largely replaced by `TIMEALL`.

If *TIMEN* is greater than 1, *TIMEX* is executed *TIMEN* times, and `TIME` prints out (number of conses)/*TIMEN*, and (computation time)/*TIMEN*. If *TIMEN*=`NIL`, it defaults to 1. This is useful for more accurate measurement on small computations.

If *TIMETYP* is 0, TIME measures and prints total *real* time as well as computation time. If *TIMETYP* = 3, TIME measures and prints garbage collection time as well as computation time. If *TIMETYP*=T, TIME measures and prints the number of pagefaults.

TIME returns the value of the last evaluation of *TIMEX*.

(BOXCOUNT *TYPE N*) [Function]

Returns the number of data objects of type *TYPE* allocated since this Interlisp system was created. *TYPE* can be any data type name (see TYPENAME, Chapter 8). If *TYPE* is NIL, it defaults to FIXP. If *N* is non-NIL, the corresponding counter is reset to *N*.

(CONSCOUNT *N*) [Function]

Returns the number of CONS cells allocated since this Interlisp system was created. If *N* is non-NIL, resets the counter to *N*. Equivalent to (BOXCOUNT 'LISTP *N*).

(PAGEFAULTS) [Function]

Returns the number of page faults since this Interlisp system was created.

BREAKDOWN

TIMEALL collects statistics for whole computations. BREAKDOWN is available to analyze the breakdown of computation time (or any other measureable quantity) function by function.

(BREAKDOWN FN_1 ... FN_N) [NLambda NoSpread Function]

The user calls BREAKDOWN giving it a list of function names (unevaluated). These functions are modified so that they keep track of various statistics.

To remove functions from those being monitored, simply UNBREAK (Chapter 15) the functions, thereby restoring them to their original state. To add functions, call BREAKDOWN on the new functions. This will not reset the counters for any functions not on the new list. However (BREAKDOWN) will zero the counters of all functions being monitored.

The procedure used for measuring is such that if one function calls other and both are "broken down", then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.

BREAKDOWN will *not* give accurate results if a function being measured is not returned from normally, e.g., a lower RETFROM (or ERROR) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

(BRKDWNRESULTS *RETURNVALUESFLG*) [Function]

BRKDWNRESULTS prints the analysis of the statistics requested as well as the number of calls to each function. If *RETURNVALUESFLG* is non-NIL, BRKDWNRESULTS will not to print the results, but instead return them in the form of a list of elements of the form (*FNNAME #CALLS VALUE*).

Example:

```
← (BREAKDOWN SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←(PRETTYDEF '(SUPERPRINT) 'FOO)
FOO.;3
←(BRKDWNRESULTS)
FUNCTIONS    TIME    #CALLS  PER CALL   %
SUPERPRINT  8.261     365    0.023     20
SUBPRINT   31.910     141    0.226     76
COMMENT1    1.612       8    0.201      4
TOTAL      41.783     514    0.081
NIL
←(BRKDWNRESULTS T)
((SUPERPRINT 365 8261) (SUBPRINT 141 31910) (COMMENT1 8
1612))
```

BREAKDOWN can be used to measure other statistics, by setting the following variables:

BRKDWNTYPE [Variable]

To use BREAKDOWN to measure other statistics, before calling BREAKDOWN, set the variable BRKDWNTYPE to the quantity of interest, e.g., TIME, CONSES, etc, or a list of such quantities. Whenever BREAKDOWN is called with BRKDWNTYPE not NIL, BREAKDOWN performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with a particular statistic, a measuring function will be defined, and the compiler will be called to compile it. The functions being broken down will be redefined to call this measuring function. When BREAKDOWN is through initializing, it sets BRKDWNTYPE back to NIL. Subsequent calls to BREAKDOWN will measure the new statistic until BRKDWNTYPE is again set and a new BREAKDOWN performed.

BRKDWNTYPES [Variable]

The list BRKDWNTYPES contains the information used to analyze new statistics. Each entry on BRKDWNTYPES should be of the form **(***TYPE FORM FUNCTION***)**, where *TYPE* is a statistic name (as would appear in BRKDWNTYPE), *FORM* computes the statistic, and *FUNCTION* (optional) converts the value of form to some more interesting quantity. For example, (TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000))) measures computation time and reports the result in seconds instead of milliseconds. BRKDWNTYPES currently contains entries for TIME, CONSES, PAGEFAULTS, BOXES, and FBOXES.

Example:

```
←(SETQ BRKDWNTYPE '(TIME CONSES))
(TIME CONSES)
←(BREAKDOWN MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←(FLIP '(A B C D E F G H C Z) '(.. $1 .. #2 ..) '(.. #3
..))
(A B D E F G H Z)
←(BRKDWNRESULTS)
FUNCTIONS  TIME    #CALLS  PER CALL   %
MATCH      0.036    1       0.036    54
CONSTRUCT  0.031    1       0.031    46
TOTAL      0.067    2       0.033
FUNCTIONS  CONSES  #CALLS  PER CALL   %
MATCH      32       1      32.000    40
CONSTRUCT  49       1      49.000    60
TOTAL      81       2      40.500
NIL
```

Occasionally, a function being analyzed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If you were using TIME, you would specify a value for *TIMEN* greater than 1 to give greater accuracy. A similar option is available for BREAKDOWN. You can specify that a function(s) be executed a multiple number of times for each measurement, and the average value reported, by including a number in the list of functions given to BREAKDOWN. For example, BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP) means normal breakdown for EDITCOM and EDIT4F but executes (the body of) EDIT4E and EQP 10 times each time they are called. Of course, the functions so measured must not cause any harmful side effects, since they are executed more than once for each call. The printout from BRKDWNRESULTS will look the same as though each function were run only once, except that the measurement will be more accurate.

Another way of obtaining more accurate measurement is to expand the call to the measuring function in-line. If the value of BRKDWNCOMPFLG is non-NIL (initially NIL), then whenever a function is broken-down, it will be redefined to call the measuring function, and then recompiled. The measuring function is expanded in-line via an appropriate macro. In addition, whenever BRKDWNTYPE is reset, the compiler is called for *all* functions for which BRKDWNCOMPFLG was set at the time they were originally broken-down, i.e. the setting of the flag at the time a function is broken-down determines whether the call to the measuring code is compiled in-line.

GAINSPACE

If you have large programs and databases, you may sometimes find yourself in a situation where you need to obtain more space, and are willing to pay the price of eliminating some or all of the context information that the various user-assistance facilities such as the programmer's assistant, file package, CLISP, etc., have accumulated during the course of his session. The function GAINSPACE provides an easy way to selectively throw away accumulated data:

(GAINSPACE) [Function]

Prints a list of deletable objects, allowing you to specify at each point what should be discarded and what should be retained. For example:

```
←(GAINSPACE)
purge history lists ? Yes
purge everything, or just the properties, e.g., SIDE,
LISPXPRINT, etc. ?
just the properties
discard definitions on property lists ? Yes
discard old values of variables ? Yes
erase properties ? No
erase CLISP translations? Yes
```

GAINSPACE is driven by the list GAINSPACEFORMS. Each element on GAINSPACEFORMS is of the form (*PRECHECK MESSAGE FORM KEYLST*). If *PRECHECK*, when evaluated, returns NIL, GAINSPACE skips to the next entry. For example, you will not be asked whether or not to purge the history list if it is not enabled. Otherwise, ASKUSER (Chapter 26) is called with the indicated *MESSAGE* and the (optional) *KEYLST*. If you respond **No**, i.e., ASKUSER returns N, GAINSPACE skips to the next entry. Otherwise, *FORM* is evaluated with the variable RESPONSE bound to the value of ASKUSER. In the above example, the *FORM* for the "**purge history lists**" question calls ASKUSER to ask "**purge everything, ...**" only if you had responded **Yes**. If you had responded with **Everything**, the second question would not have been asked.

The "**erase properties**" question is driven by a list SMASHPROPSMENU. Each element on this list is of the form (*MESSAGE . PROPS*). You are prompted with *MESSAGE* (by ASKUSER), and if your response is **Yes**, *PROPS* is added to the list SMASHPROPS. The "**discard definitions on property lists**" and "**discard old values of variables**" questions also add to SMASHPROPS. You will not be prompted for any entry on SMASHPROPSMENU for which all of the corresponding properties are already on SMASHPROPS. SMASHPROPS is initially set to the value of SMASHPROPSLST. This permits you to specify in advance those properties which you always want discarded, and not be asked about them subsequently. After finishing all the entries on GAINSPACEFORMS, GAINSPACE checks to see if the value of SMASHPROPS is non-NIL, and if so, does a MAPATOMS, i.e., looks at every atom in the system, and erases the indicated properties.

You can change or add new entries to GAINSPACEFORMS or SMASHPROPSMENU, so that GAINSPACE can also be used to purge structures that your programs have accumulated.

Using Data Types Instead of Records

If a program uses large numbers of large data structures, there are several advantages to representing them as user data types rather than as list structures. The primary advantage is increased speed: accessing and setting the fields of a data type can be significantly faster than walking through a list with repeated CARs and CDRs. Also, compiled code for referencing data types is usually smaller. Finally, by reducing the number of objects created (one object against many list cells), this can reduce the expense of garbage collection.

User data types are declared by using the DATATYPE record type (Chapter 8). If a list structure has been defined using the RECORD record type (Chapter 8), and all accessing operations are written using the record package's **fetch**, **replace**, and **create** operations, changing from RECORDs to DATATYPEs only requires editing the record declaration (using EDITREC, Chapter 8) to replace declaration type RECORD by DATATYPE, and recompiling.

Note: There are some minor disadvantages with allocating new data types: First, there is an upper limit on the number of data types which can exist. Also, space for data types is allocated a page at a time, so each data type has at least one page assigned to it, which may be wasteful of space if there are only a few examples of a given data type. These problems should not effect most applications programs.

Using Incomplete File Names

Currently, Interlisp allows you to specify an open file by giving the file name. If the file name is incomplete (it doesn't have the device/host, directory, name, extension, and version number all supplied), the system converts it to a complete file name, by supplying defaults and searching through directories (which may be on remote file servers), and then searches the open streams for one corresponding to that file name. This file name-completion process happens whenever any I/O function is given an incomplete file name, which can cause a serious performance problem if I/O operations are done repeatedly. In general, it is much faster to convert an incomplete file name to a stream once, and use the stream from then on. For example, suppose a file is opened with `(SETQ STRM (OPENSTREAM 'MYNAME 'INPUT))`. After doing this, `(READC 'MYNAME)` and `(READC STRM)` would both work, but `(READC 'MYNAME)` would take longer (sometimes orders of magnitude longer). This could seriously effect the performance if a program which is doing many I/O operations.

At some point in the future, when multiple streams are supported to a single file, the feature of mapping file names to streams will be removed. This is yet another reason why programs should use streams as handles to open files, instead of file names.

For more information on efficiency considerations when using files, see Chapter 24.

Using "Fast" and "Destructive" Functions

Among the functions used for manipulating objects of various data types, there are a number of functions which have "fast" and "destructive" versions. You should be aware of what these functions do, and when they should be used.

"Fast" functions: By convention, a function named by prefixing an existing function name with `F` indicates that the new function is a "fast" version of the old. These usually have the same definitions as the slower versions, but they compile open and run without any "safety" error checks. For example, `FNTH` runs faster than `NTH`, however, it does not make as many checks (for lists ending with anything but `NIL`, etc). If these functions are given arguments that are not in the form that they expect, their behavior is unpredictable; they may run forever, or cause a system error. In general, you should only use "fast" functions in code that has already been completely debugged, to speed it up.

"Destructive" functions: By convention, a function named by prefixing an existing function with `D` indicates the new function is a "destructive" version of the old one, which does not make any new structure but cannibalizes its argument(s). For example, `REMOVE` returns a copy of a list with a particular element removed, but `DREMOVE` actually changes the list structure of the list. (Unfortunately, not all destructive functions follow this naming convention: the destructive version of `APPEND` is `NCONC`.) You should be careful when using destructive functions that they do not inadvertantly change data structures.

22. PROCESSES

The Medley Process mechanism provides an environment in which multiple Lisp processes can run in parallel. Each executes in its own stack space, but all share a global address space. The current process implementation is cooperative; i.e., process switches happen voluntarily, either when the process in control has nothing to do or when it is in a convenient place to pause. There is no preemption or guaranteed service, so you cannot run something demanding (e.g., Chat) at the same time as something that runs for long periods without yielding control. Keyboard input and network operations block with great frequency, so processes currently work best for highly interactive tasks (editing, making remote files).

In Medley, the process mechanism is already turned on, and is expected to stay on during normal operations, as some system facilities (in particular, most network operations) require it. However, under exceptional conditions, the following function can be used to turn the world off and on:

(**PROCESSWORLD** *FLG*) [Function]

Starts up the process world, or if *FLG* = `OFF`, kills all processes and turns it off. Normally does not return. The environment starts out with two processes: a top-level `EVALQT` (the initial "tty" process) and the "background" process, which runs the window mouse handler and other system background tasks.

`PROCESSWORLD` is intended to be called at the top level of Interlisp, not from within a program. It does not toggle some sort of switch; rather, it constructs some new processes in a new part of the stack, leaving any callers of `PROCESSWORLD` in a now inaccessible part of the stack. Calling `(PROCESSWORLD 'OFF)` is the only way the call to `PROCESSWORLD` ever returns.

(**HARDRESET**) [Function]

Resets the whole world, and rebuilds the stack from scratch. This is "harder" than doing `RESET` to every process, because it also resets system internal processes (such as the keyboard handler).

`HARDRESET` automatically turns the process world on (or resets it if it was on), unless the variable `AUTOPROCESSFLG` is `NIL`.

Creating and Destroying Processes

(**ADD.PROCESS** *FORM* $PROP_1$ $VALUE_1$ ... $PROP_N$ $VALUE_N$) [NoSpread Function]

Creates a new process evaluating *FORM*, and returns its process handle. The process's stack environment is the top level, i.e., the new process does not have access to the environment in which `ADD.PROCESS` was called; all such information must be passed as arguments in *FORM*. The process runs until *FORM* returns or the process is explicitly deleted. An untrapped error within the process also deletes the process (unless its `RESTARTABLE` property is `T`), in which case a message is printed to that effect.

The remaining arguments are alternately property names and values. Any property/value pairs acceptable to PROCESSPROP may be given, but the following two are directly relevant to ADD.PROCESS:

NAME Value can be a symbol or a string; if not given, the process name is taken from (CAR *FORM*). ADD.PROCESS may pack the name with a number to make it unique. Process names are treated as case-insensitive strings. This name is solely for the convenience of manipulating processes at Lisp type-in; e.g., the name can be given as the *PROC* argument to most process functions, and the name appears in menus of processes. However, programs should normally only deal in process handles, both for efficiency and to avoid the confusion that can result if two processes have the same defining form.

SUSPEND If the value is non-NIL, the new process is created but then immediately suspended; i.e., the process does not actually run until woken by a WAKE.PROCESS (below).

(**PROCESSPROP** *PROC PROP NEWVALUE*) [NoSpread Function]

Used to get or set the values of certain properties of process *PROC*, in a manner analogous to WINDOWPROP. If *NEWVALUE* is supplied (including if it is NIL), property *PROP* is given that value. In all cases, returns the old value of the property. The following properties have special meaning for processes; all others are uninterpreted:

NAME Value is a symbol used for identifying the process to the user.

FORM Value is the Lisp form used to start the process (readonly).

RESTARTABLE Value is a flag indicating the disposition of the process following errors or hard resets:

NIL or NO (the default): If an untrapped error (or Control-E or Control-D) causes its form to be exited, the process is deleted. The process is also deleted if a HARDRESET (or Control-D from RAID) occurs, causing the entire Process world to be reinitialized.

T or YES: The process is automatically restarted on errors or HARDRESET. This is the normal setting for persistent "background" processes, such as the mouse process, that can safely restart themselves on errors.

HARDRESET: The process is deleted as usual if an error causes its form to be exited, but it *is* restarted on a HARDRESET. This setting is preferred for persistent processes for which an error is an unusual condition, one that might repeat itself if the process were simply blindly restarted.

RESTARTFORM If the value is non-NIL, it is the form used if the process is restarted (instead of the value of the FORM property). Of course, the process must also have a non-NIL RESTARTABLE prop for this to have any effect.

BEFOREEXIT If the value is the atom DON'T, it will not be interrupted by a LOGOUT. If LOGOUT is attempted before the process finishes, a message will appear saying that Interlisp is waiting for the process to finish. If you want the LOGOUT to proceed without waiting, you must use the process status window (from the background menu) to delete the process.

AFTEREXIT Value indicates the disposition of the process following a resumption of Lisp after some exit (LOGOUT, SYSOUT, MAKESYS). Possible values are:

DELETE: Delete the process.

SUSPEND: Suspend the process; i.e., do not let it run until it is explicitly woken.

An event: Cause the process to be suspended waiting for the event (See the Events section below).

INFOHOOK Value is a function or form used to provide information about the process, in conjunction with the INFO command in the process status window (see the Process Status Window section below).

WINDOW Value is a window associated with the process, the process's "main" window. Used to switch the tty process to this process when you click in this window (see the Switching the TTY Process section below).

Setting the WINDOW property does not set the primary I/O stream (NIL) or the terminal I/O stream (T) to the window. When a process is created, I/O operations to the NIL or T stream will cause a new window to appear. TTYDISPLAYSTREAM (see Chapter 26) should be used to set the terminal I/O stream of a process to a specific window.

TTYENTRYFN Value is a function that is applied to the process when the process is made the tty process (see the Switching the TTY Process section below).

TTYEXITFN Value is a function that is applied to the process when the process ceases to be the tty process (see the Switching the TTY Process section below).

(**THIS.PROCESS**) [Function]

Returns the handle of the currently running process, or NIL if the Process world is turned off.

(**DEL.PROCESS** *PROC*) [Function]

Deletes process *PROC*. *PROC* may be a process handle (returned by ADD.PROCESS), or its name. If *PROC* is the currently running process, DEL.PROCESS does not return!

(**PROCESS.RETURN** *VALUE*) [Function]

Terminates the currently running process, causing it to "return" *VALUE*. There is an implicit PROCESS.RETURN around the *FORM* argument given to ADD.PROCESS, so that normally a process can finish by simply returning; PROCESS.RETURN is supplied for earlier termination.

(**PROCESS.RESULT** *PROCESS WAITFORRESULT*) [Function]

If *PROCESS* has terminated, returns the value, if any, that it returned. This is either the value of a PROCESS.RETURN or the value returned from the form given to ADD.PROCESS. If the process was aborted, the value is NIL. If *WAITFORRESULT* is true, PROCESS.RESULT blocks until *PROCESS* finishes, if necessary; otherwise, it returns NIL immediately if *PROCESS* is still running. *PROCESS* must be the actual process handle returned from ADD.PROCESS, not a process name, as the association between handle and name disappears when the process finishes (and the process handle itself is then garbage collected if no one else has a pointer to it).

(**PROCESS.FINISHEDP** *PROCESS*) [Function]

True if *PROCESS* has terminated. The value returned is an indication of how it finished: NORMAL or ERROR.

(**PROCESSP** *PROC*) [Function]

True if *PROC* is the handle of an active process, i.e., one that has not yet finished.

(**RELPROCESSP** *PROCHANDLE*) [Function]

True if *PROCHANDLE* is the handle of a deleted process. This is analogous to RELSTKP. It differs from PROCESS.FINISHEDP in that it never causes an error, while PROCESS.FINISHEDP can cause an error if its *PROC* argument is not a process at all.

(**RESTART.PROCESS** *PROC*) [Function]

Unwinds *PROC* to its top level and reevaluates its form. This is effectively a DEL.PROCESS followed by the original ADD.PROCESS.

(**MAP.PROCESSES** *MAPFN*) [Function]

Maps over all processes, calling *MAPFN* with three arguments: the process handle, its name, and its form.

(**FIND.PROCESS** *PROC ERRORFLG*) [Function]

If *PROC* is a process handle or the name of a process, returns the process handle for it, else NIL. If *ERRORFLG* is T, generates an error if *PROC* is not, and does not name, a live process.

Process Control Constructs

(**BLOCK** *MSECSWAIT TIMER*) [Function]

Yields control to the next waiting process, assuming any is ready to run. If *MSECSWAIT* is specified, it is a number of milliseconds to wait before returning, or T, meaning wait forever (until explicitly woken). Alternatively, *TIMER* can be given as a millisecond timer (as returned by SETUPTIMER, Chapter 12) of an absolute time at which to wake up. In any of those cases, the process enters the *waiting* state until the time limit is up. BLOCK with no arguments leaves the process in the *runnable* state, i.e., it returns as soon as every other runnable process of the same priority has had a chance.

BLOCK can be aborted by interrupts such as Control-D, Control-E, or Control-B. BLOCK will return before its timeout is completed, if the process is woken by WAKE.PROCESS, PROCESS.EVAL, or PROCESS.APPLY.

(**DISMISS** *MSECSWAIT TIMER NOBLOCK*) [Function]

DISMISS is used to dismiss the current process for a given period of time. Similar to BLOCK, except that:

- DISMISS is guaranteed not to return until the specified time has elapsed
- *MSECSWAIT* cannot be T to wait forever
- If *NOBLOCK* is T, DISMISS will not allow other processes to run, but will busy-wait until the amount of time given has elapsed.

(**WAKE.PROCESS** *PROC STATUS*) [Function]

Explicitly wakes process *PROC*, i.e., makes it *runnable*, and causes its call to BLOCK (or other waiting function) to return *STATUS*. This is one simple way to notify a process of some happening; however, note that if WAKE.PROCESS is applied to a process more than once before the process actually gets its turn to run, it sees only the latest *STATUS*.

(**SUSPEND.PROCESS** *PROC*) [Function]

Blocks process *PROC* indefinitely, i.e., *PROC* will not run until it is woken by a WAKE.PROCESS.

The following three functions allow access to the stack context of some other process. They require a little bit of care, and are computationally non-trivial, but they do provide a more powerful way of manipulating another process than WAKE.PROCESS allows.

(**PROCESS.EVALV** *PROC VAR*) [Function]

Performs (EVALV *VAR*) in the stack context of *PROC*.

(**PROCESS.EVAL** *PROC FORM WAITFORRESULT*) [Function]

Evaluates *FORM* in the stack context of *PROC*. If *WAITFORRESULT* is true, blocks until the evaluation returns a result, else allows the current process to run in parallel with the evaluation. Any errors that occur will be in the context of *PROC*, so be careful. In particular, note that

```
(PROCESS.EVAL PROC '(NLSETQ (FOO)))
```

and

```
(NLSETQ (PROCESS.EVAL PROC '(FOO)))
```

behave quite differently if FOO causes an error. And it is quite permissible to intentionally cause an error in proc by performing

```
(PROCESS.EVAL PROC'(ERROR!))
```

If *WAITFORRESULT* is true and the computation in the other process aborts or the other process is killed PROCESS.EVAL returns :ABORTED

After *FORM* is evaluated in *PROC*, the process *PROC* is woken up, even if it was running BLOCK or AWAIT.EVENT. This is necessary because an event of interest may have occurred while the process was evaluating *FORM*.

(**PROCESS.APPLY** *PROC FN ARGS WAITFORRESULT*) [Function]

Performs (APPLY *FN ARGS*) in the stack context of *PROC*. Note the same warnings as with PROCESS.EVAL.

Events

An "event" is a synchronizing primitive used to coordinate related processes, typically producers and consumers. Consumer processes can "wait" on events, and producers "notify" events.

(**CREATE.EVENT** *NAME*) [Function]

Returns an instance of the EVENT datatype, to be used as the event argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(**AWAIT.EVENT** *EVENT TIMEOUT TIMERP*) [Function]

Suspends the current process until *EVENT* is notified, or until a timeout occurs. If *TIMEOUT* is NIL, there is no timeout. Otherwise, timeout is either a number of milliseconds to wait, or, if *TIMERP* is T, a millisecond timer set to expire at the desired time using SETUPTIMER (see Chapter 12).

(**NOTIFY.EVENT** *EVENT ONCEONLY*) [Function]

If there are processes waiting for *EVENT* to occur, causes those processes to be placed in the running state, with *EVENT* returned as the value from AWAIT.EVENT. If *ONCEONLY* is true, only runs the first process waiting for the event (this should only be done if the programmer knows that there can only be one process capable of responding to the event at once).

The meaning of an event is up to the programmer. In general, however, the notification of an event is merely a hint that something of interest to the waiting process has happened; the process should still verify that the conceptual event actually occurred. That is, *the process should be written so that it operates correctly even if woken up before the timeout and in the absence of the notified event.* In particular, the completion of PROCESS.EVAL and related operations in effect wakes up the process in which they were performed, since there is no secure way of knowing whether the event of interest occurred while the process was busy performing the PROCESS.EVAL.

There is currently one class of system-defined events, used with the network code. Each Pup and NS socket has associated with it an event that is notified when a packet arrives on the socket; the event can be obtained by calling PUPSOCKETEVENT or NSOCKETEVENT, respectively (see Chapter 30).

Monitors

It is often the case that cooperating processes perform operations on shared structures, and some mechanism is needed to prevent more than one process from altering the structure at the same time. Some languages have a construct called a monitor, a collection of functions that access a common structure with mutual exclusion provided and enforced by the compiler via the use of monitor locks. Medley has taken this implementation notion as the basis for a mutual exclusion capability suitable for a dynamically-scoped environment.

A monitorlock is an object created by you and associated with (e.g., stored in) some shared structure that is to be protected from simultaneous access. To access the structure, a program waits for the lock to be free, then takes ownership of the lock, accesses the structure, then releases the lock. The functions and macros below are used:

(**CREATE.MONITORLOCK** *NAME*) [Function]

Returns an instance of the MONITORLOCK datatype, to be used as the lock argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(**WITH.MONITOR** *LOCK* $FORM_1$ ... $FORM_N$) [Macro]

Evaluates $FORM_1$ ... $FORM_N$ while owning *LOCK*, and returns the value of $FORM_N$. This construct is implemented so that the lock is released even if the form is exited via error (currently implemented with RESETLST).

Ownership of a lock is dynamically scoped: if the current process already owns the lock (e.g., if the caller was itself inside a WITH.MONITOR for this lock), WITH.MONITOR does not wait for the lock to be free before evaluating $FORM_1$ ... $FORM_N$.

(**WITH.FAST.MONITOR** *LOCK* $FORM_1$ ... $FORM_N$) [Macro]

Like WITH.MONITOR, but implemented without the RESETLST. User interrupts (e.g., Control-E) are inhibited during the evaluation of $FORM_1$ ... $FORM_N$.

Programming restriction: the evaluation of $FORM_1$ ... $FORM_N$ must not error (the lock would not be released). This construct is mainly useful when the forms perform a small, safe computation that never errors and need never be interrupted.

(**MONITOR.AWAIT.EVENT** *RELEASELOCK EVENT TIMEOUT TIMERP*) [Function]

For use in blocking inside a monitor. Performs (AWAIT.EVENT *EVENT TIMEOUT TIMERP*), but releases *RELEASELOCK* first, and reobtains the lock (possibly waiting) on wakeup.

Typical use for MONITOR.AWAIT.EVENT: A function wants to perform some operation on *FOO*, but only if it is in a certain state. It has to obtain the lock on the structure to make sure that the state of the structure does not change between the time it tests the state and performs the operation. If the state turns out to be bad, it then waits for some other process to make the state good, meanwhile releasing the lock so that the other process can alter the structure.

```
(WITH.MONITOR FOO-LOCK
  (until CONDITION-OF-FOO
  do (MONITOR.AWAIT.EVENT FOO-LOCK EVENT-FOO-
CHANGED TIMEOUT))
    OPERATE-ON-FOO)
```

It is sometimes convenient for a process to have WITH.MONITOR at its top level and then do all its interesting waiting using MONITOR.AWAIT.EVENT. Not only is this often cleaner, but in the present implementation in cases where the lock is frequently accessed, it saves the RESETLST overhead of WITH.MONITOR.

Programming restriction: There must not be an ERRORSET between the enclosing WITH.MONITOR and the call to MONITOR.AWAIT.EVENT such that the ERRORSET would catch an ERROR! and continue inside the monitor, for the lock would not have been reobtained. (The reason for this restriction is that, although MONITOR.AWAIT.EVENT won't itself error, you could have caused an error with an interrupt, or a PROCESS.EVAL in the context of the waiting process that produced an error.)

On rare occasions it may be useful to manipulate monitor locks directly. The following two functions are used in the implementation of WITH.MONITOR:

(**OBTAIN.MONITORLOCK** *LOCK DONTWAIT UNWINDSAVE*) [Function]

Takes possession of *LOCK*, waiting if necessary until it is free, unless *DONTWAIT* is true, in which case it returns NIL immediately. If *UNWINDSAVE* is true, performs a RESETSAVE to be unwound when the enclosing RESETLST exits. Returns *LOCK* if *LOCK* was successfully obtained, T if the current process already owned *LOCK*.

(**RELEASE.MONITORLOCK** *LOCK EVENIFNOTMINE*) [Function]

Releases *LOCK* if it is owned by the current process, and wakes up the next process, if any, waiting to obtain the lock.

If *EVENIFNOTMINE* is non-NIL, the lock is released even if it is not owned by the current process.

When a process is deleted, any locks it owns are released.

Global Resources

The biggest source of problems in the multi-processing environment is the matter of global resources. Two processes cannot both use the same global resource if there can be a process switch in the middle of their use (currently this means calls to BLOCK, but ultimately with a preemptive scheduler means anytime). Thus, user code should be wary of its own use of global variables, if it ever makes sense for the code to be run in more than one process at a time. "State" variables private to a process should generally be bound in that process; structures that are shared among processes (or resources used privately but expensive to duplicate per process) should be protected with monitor locks or some other form of synchronization.

Aside from user code, however, there are many *system* global variables and resources. Most of these arise historically from the single-process Interlisp-10 environment, and will eventually be changed in Medley to behave appropriately in a multi-processing environment. Some have already been changed, and are described below. Two other resources not generally thought of as global variables—the keyboard and the mouse—are particularly idosyncratic, and are discussed in the next section.

are allocated per process in Medley: primary input and output (the streams affected by INPUT and OUTPUT), terminal input and output (the streams designated by the name T), the primary read table and primary terminal table, and dribble files. Thus, each process can print to its own primary output, print to the terminal, read from a different primary input, all without interfering with another process's reading and printing.

Each process begins life with its primary and terminal input/output streams set to a dummy stream. If the process attempts input or output using any of those dummy streams, e.g., by calling (READ T), or (PRINT & T), a tty window is automatically created for the process, and that window becomes the primary input/output and terminal input/output for the process. The default tty window is created at or near the region specified in the variable DEFAULTTTYREGION.

A process can, of course, call TTYDISPLAYSTREAM explicitly to give itself a tty window of its own choosing, in which case the automatic mechanism never comes into play. Calling TTYDISPLAYSTREAM when a process has no tty window not only sets the terminal streams, but also sets the primary input and output streams to be that window, assuming they were still set to the dummy streams.

(**HASTTYWINDOWP** *PROCESS*) [Function]

Returns T if the process *PROCESS* has a tty window; NIL otherwise. If *PROCESS* is NIL, it defaults to the current process.

Other system resources that are typically changed by RESETFORM, RESETLST, or RESETVARS are all global entities. In the multiprocessing environment, these constructs are suspect, as there is no provision for "undoing" them when a process switch occurs. For example, in the current release of Medley, it is not possible to set the print radix to 8 inside only one process, as the print radix is a global entity.

Note that RESETFORM and similar expressions are perfectly valid in the process world, and even quite useful, when they manipulate things strictly within one process. The process world is arranged so that deleting a process also unwinds any RESETxxx expressions that were performed in the process and are still waiting to be unwound, exactly as if a Control-D had reset the process to the top. Additionally, there is an implicit RESETLST at the top of each process, so that RESETSAVE can be used as a way of providing "cleanup" functions for when a process is deleted. For these, the value of RESETSTATE (see Chapter 14) is NIL if the process finished normally, ERROR if it was aborted by an error, RESET if the process was explicitly deleted, and HARDRESET if the process is being restarted after a HARDRESET or a RESTART.PROCESS.

Typein and the TTY Process

There is one global resource, the keyboard, that is particularly problematic to share among processes. Consider, for example, having two processes both performing (READ T). Since the keyboard input routines block while there is no input, both processes would spend most of their time blocking, and it would simply be a matter of chance which process received each character of type-in.

To resolve such dilemmas, the system designates a distinguished process, termed the *tty process*, that is assumed to be the process that is involved in terminal interaction. Any type-in from the keyboard goes to that process. If a process other than the tty process requests keyboard input, it blocks until it becomes the tty process. When the tty process is switched (in any of the ways described further below), any typeahead that occurred before the switch is saved and associated with the current tty process. Thus, it is always the case that keystrokes are sent to the process that is the tty process at the time of the keystrokes, regardless of when that process actually gets around to reading them.

BACKGROUNDFNS [Variable]

A list of functions to call "in the background". The system runs a process (called "BACKGROUND") whose sole task is to call each of the functions on the list BACKGROUNDFNS repeatedly. Each element is the name of a function of no arguments. This is a good place to put cheap background tasks that only do something once in a while and hence do not want to spend their own separate process on it. However, note that it is considered good citizenship for a background function with a time-consuming task to spawn a separate process to do it, so that the other background functions are not delayed.

TTYBACKGROUNDFNS [Variable]

This list is like BACKGROUNDFNS, but the functions are only called while in a tty input wait. That is, they always run in the tty process, and only when the user is not actively typing. For example, the flashing caret is implemented by a function on this list. Again, functions on this list should spend very little time (much less than a second), or else spawn a separate process.

It is less immediately obvious how to handle keyboard interrupt characters, as their action is asynchronous and not always tied to type-in. Interrupt handling is described in the Handling of Interrupts section below.

Switching the TTY Process

Any process can make itself be the tty process by calling TTY.PROCESS.

(**TTY.PROCESS** *PROC*) [Function]

Returns the handle of the current tty process. In addition, if *PROC* is non-NIL, makes it be the tty process. The special case of *PROC* = T is interpreted to mean the executive process; this is sometimes useful when a process wants to explicitly give up being the tty process.

(**TTY.PROCESSP** *PROC*) [Function]

True if *PROC* is the tty process; *PROC* defaults to the running process. Thus, (TTY.PROCESSP) is true if the caller is the tty process.

(**WAIT.FOR.TTY** *MSECS NEEDWINDOW*) [Function]

Efficiently waits until (TTY.PROCESSP) is true. WAIT.FOR.TTY is called internally by the system functions that read from the terminal; user code thus need only call it in special cases.

If *MSECS* is non-NIL, it is the number of milliseconds to wait before timing out. If WAIT.FOR.TTY times out before (TTY.PROCESSP) is true, it returns NIL, otherwise it returns T. If *MSECS* is NIL, WAIT.FOR.TTY will not time out.

If *NEEDWINDOW* is non-NIL, WAIT.FOR.TTY opens a TTY window for the current process if one isn't already open.

WAIT.FOR.TTY spawns a new mouse process if called under the mouse process (see SPAWN.MOUSE, in the Keeping the Mouse Alive section below).

In some cases, such as in functions invoked as a result of mouse action or a user's typed-in call, it is reasonable for the function to invoke TTY.PROCESS itself so that it can take subsequent user type in. In other cases, however, this is too undisciplined; it is desirable to let you designate which process type-in should be directed to. This is most conveniently done by mouse action.

The system supports the model that "to type to a process, you click in its window." To cooperate with this model, any process desiring keyboard input should put its process handle as the PROCESS property of its window(s). To handle the common case, the function TTYDISPLAYSTREAM does this automatically when the ttydisplaystream is switched to a new window. A process can own any number of windows; clicking in any of those windows gives the process the tty.

This mechanism suffices for most casual process writers. For example, if a process wants all its input/output interaction to occur in a particular window that it has created, it should just make that window be its tty window by calling TTYDISPLAYSTREAM. Thereafter, it can PRINT or READ to/from the T stream; if the process is not the tty process at the time that it calls READ, it will block until the user clicks in the window.

For those needing tighter control over the tty, the default behavior can be overridden or supplemented. The remainder of this section describes the mechanisms involved.

There is a window property WINDOWENTRYFN that controls whether and how to switch the tty to the process owning a window. The mouse handler, before invoking any normal BUTTONEVENTFN, specifically notices the case of a button going down in a window that belongs to a process (i.e., has a PROCESS window property) that is not the tty process. In this case, it invokes the window's WINDOWENTRYFN of one argument (WINDOW). WINDOWENTRYFN defaults to GIVE.TTY.PROCESS:

(**GIVE.TTY.PROCESS** *WINDOW*) [Function]

If *WINDOW* has a PROCESS property, performs (TTY.PROCESS (WINDOWPROP *WINDOW* 'PROCESS)) and then invokes *WINDOW*'s BUTTONEVENTFN function (or RIGHTBUTTONFN if the right button is down).

There are some cases where clicking in a window does not always imply that the user wants to talk to that window. For example, clicking in a text editor window with a shift key held down means to "shift-select" some piece of text into the input buffer of the *current* tty process. The editor supports this by supplying a WINDOWENTRYFN that performs GIVE.TTY.PROCESS if no shift key is down, but goes into its shift-select mode, without changing the tty process, if a shift key is down. The shift-select mode performs a BKSYSBUF of the selected text when the shift key is let up, the BKSYSBUF feeding input to the current tty process.

Sometimes a process wants to be notified when it becomes the tty process, or stops being the tty process. To support this, there are two process properties, TTYEXITFN and TTYENTRYFN. The actions taken by TTY.PROCESS when it switches the tty to a new process are as follows: the former tty process's TTYEXITFN is called with two arguments (*OLDTTYPROCESS NEWTTYPROCESS*); the new process is made the tty process; finally, the new tty process's TTYENTRYFN is called with two arguments (*NEWTTYPROCESS OLDTTYPROCESS*). Normally the TTYENTRYFN and TTYEXITFN need only their first argument, but the other process involved in the switch is supplied for completeness. In the present system, most processes want to interpret the keyboard in the same way, so it is considered the responsibility of any process that changes the keyboard interpretation to restore it to the normal state by its TTYEXITFN.

A window is "owned" by the last process that anyone gave as the window's PROCESS property. Ordinarily there is no conflict here, as processes tend to own disjoint sets of windows (though, of course, cooperating processes can certainly try to confuse each other). The only likely problem arises with that most global of windows, PROMPTWINDOW. Programs should not be tempted to read from PROMPTWINDOW. This is not usually necessary anyway, as the first attempt to read from T in a process that has not set its TTYDISPLAYSTREAM to its own window causes a tty window to be created for the process (see the Global Resources section above).

Handling of Interrupts

At the time that a keyboard interrupt character (see Chapter 29) is struck, any process could be running, and some decision must be made as to which process to actually interrupt. To the extent that keyboard interrupts are related to type-in, most interrupts are taken in the tty process; however, the following are handled specially:

RESET (initially Control-D)
ERROR (initially Control-E)

These interrupts are taken in the mouse process, if the mouse is not in its idle state; otherwise they are taken in the tty process. Thus, Control-E can be used to abort some mouse-invoked window action, such as the Shape command. As a consequence, note that if the mouse invokes some lengthy computation that the user thinks of as "background", Control-E still aborts it, even though that may not have been what the user intended. Such lengthy computations, for various reasons, should generally be performed by spawning a separate process to perform them. The RESET interrupt in a process other than the executive is interpreted exactly as if an error unwound the process to its top level: if the process was designated RESTARTABLE = T, it is restarted; otherwise it is killed.

HELP (initially Control-G) A menu of processes is presented to the user, who is asked to select which one the interrupt should occur in. The current tty process appears with a * next to its name at the top of the menu. The menu also includes an entry "[Spawn Mouse]", for the common case of needing a mouse because the mouse process is currently tied up running someone's BUTTONEVENTFN; selecting this entry spawns a new mouse process, and no break occurs.

BREAK (initially Control-B) Performs the HELP interrupt in the mouse process, if the mouse is not in its idle state; otherwise it is performed in the tty process.

RUBOUT (initially DELETE) This interrupt clears typeahead in *all* processes.

RAID, STACK OVERFLOW STORAGE FULL These interrupts always occur in whatever process was running at the time the interrupt struck. In the cases of STACK OVERFLOW and STORAGE FULL, this means that the interrupt is more likely to strike in the offending process (especially if it is a "runaway" process that is not blocking). Note, however, that this process is still not necessarily the guilty party; it could be an innocent bystander that just happened to use up the last of a resource prodigiously consumed by some other process.

Keeping the Mouse Alive

Since the window mouse handler runs in its own process, it is not available while a window's BUTTONEVENTFN function (or any of the other window functions invoked by mouse action) is running. This leads to two sorts of problems: (1) a long computation underneath a BUTTONEVENTFN deprives the user of the mouse for other purposes, and (2) code that runs as a BUTTONEVENTFN cannot rely on other BUTTONEVENTFNs running, which means that there some pieces of code that run differently from normal when run under the mouse process. These problems are addressed by the following functions:

(**SPAWN.MOUSE** —) [Function]

Spawns another mouse process, allowing the mouse to run even if it is currently "tied up" under the current mouse process. This function is intended mainly to be typed in at the Lisp executive when you notice the mouse is busy.

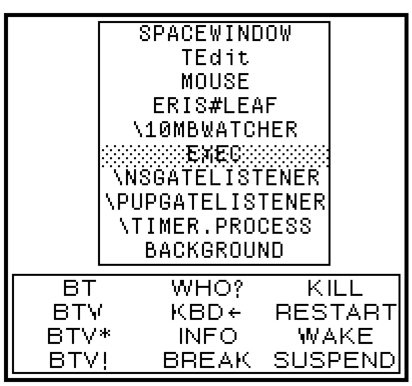
(**ALLOW.BUTTON.EVENTS**) [Function]

Performs a (SPAWN.MOUSE) only when called underneath the mouse process. This should be called (once, on entry) by any function that relies on BUTTONEVENTFNs for completion, if there is any possibility that the function will itself be invoked by a mouse function.

It never hurts, at least logically, to call SPAWN.MOUSE or ALLOW.BUTTON.EVENTS needlessly, as the mouse process arranges to quietly kill itself if it returns from the user's BUTTONEVENTFN and finds that another mouse process has sprung up in the meantime. (There is, of course, some computational expense.)

Process Status Window

The background menu command PSW (see Chapter 27) and the function PROCESS.STATUS.WINDOW (below) create a "Process Status Window", that allows you to examine and manipulate all of the existing processes:



The window consists of two menus. The top menu lists all the processes at the moment. Commands in the bottom menu operate on the process selected in the top menu (EXEC in the example above). The commands are:

BT, **BTV**, **BTV***, **BTV!** Displays a backtrace of the selected process.

WHO? Changes the selection to the tty process, i.e., the one currently in control of the keyboard.

KBD← Associates the keyboard with the selected process; i.e., makes the selected process be the tty process.

INFO If the selected process has an INFOHOOK property, calls it. The hook may be a function, which is then applied to two arguments, the process and the button (LEFT or MIDDLE) used to invoke INFO, or a form, which is simply EVAL'ed. The APPLY or EVAL happens in the context of the selected process, using PROCESS.APPLY or PROCESS.EVAL. The INFOHOOK process property can be set using PROCESSPROP (see the Creating and Destroying Processes section above).

BREAK Enter a break under the selected process. This has the side effect of waking the process with the value returned from the break.

KILL Deletes the selected process.

RESTART Restarts the selected process.

WAKE Wakes the selected process. Prompts for a value to wake it with (see WAKE.PROCESS).

SUSPEND Suspends the selected process; i.e., causes it to block indefinitely (until explicitly woken).

(**PROCESS.STATUS.WINDOW** *WHERE*) [Function]

Puts up a process status window that provides several debugging commands for manipulating running processes. If the window is already up, PROCESS.STATUS.WINDOW refreshes it. If *WHERE* is a position, the window is placed in that position; otherwise, you are prompted for a position.

Currently, the process status window runs under the mouse process, like other menus, so if the mouse is unavailable (e.g., a mouse function is performing an extensive computation), you may be unable to use the process status window (you can try SPAWN.MOUSE, of course).

Non-Process Compatibility

This section describes some considerations for authors of programs that ran in the old single-process Medley environment, and now want to make sure they run properly in the multi-processing world. The biggest problem to watch out for is code that runs underneath the mouse handler. Writers of mouse handler functions should remember that in the process world the mouse handler runs in its own process, and hence (a) you cannot depend on finding information on the stack (stash it in the window instead), and (b) while your function is running, the mouse is not available (if you have any non-trivial computation to do, spawn a process to do it, notify one of your existing processes to do it, or use PROCESS.EVAL to run it under some other process).

The following functions are meaningful even if the process world is not on: BLOCK (invokes the system background routine, which includes handling the mouse); TTY.PROCESS, THIS.PROCESS (both return NIL); and TTY.PROCESSP (returns T, i.e., anyone is allowed to take tty input). In addition, the following two functions exist in both worlds:

(**EVAL.AS.PROCESS** *FORM*) [Function]

Same as (ADD.PROCESS *FORM* 'RESTARTABLE 'NO), when processes are running, EVAL when not. This is highly recommended for mouse functions that perform any non-trivial activity.

(**EVAL.IN.TTY.PROCESS** *FORM WAITFORRESULT*) [Function]

Same as (PROCESS.EVAL (TTY.PROCESS) *FORM WAITFORRESULT*), when processes are running, EVAL when not.

Most of the process functions that do not take a process argument can be called even if processes aren't running. ADD.PROCESS creates, but does not run, a new process (it runs when PROCESSWORLD is called).

23. PROCESSES

The Interlisp-D Process mechanism provides an environment in which multiple Lisp processes can run in parallel. Each executes in its own stack space, but all share a global address space. The current process implementation is cooperative; i.e., process switches happen voluntarily, either when the process in control has nothing to do or when it is in a convenient place to pause. There is no preemption or guaranteed service, so you cannot run something demanding (e.g., Chat) at the same time as something that runs for long periods without yielding control. Keyboard input and network operations block with great frequency, so processes currently work best for highly interactive tasks (editing, making remote files).

In Interlisp-D, the process mechanism is already turned on, and is expected to stay on during normal operations, as some system facilities (in particular, most network operations) require it. However, under exceptional conditions, the following function can be used to turn the world off and on:

`(PROCESSWORLD` *FLG*`)` [Function]

> Starts up the process world, or if *FLG* = `OFF`, kills all processes and turns it off. Normally does not return. The environment starts out with two processes: a top-level `EVALQT` (the initial "tty" process) and the "background" process, which runs the window mouse handler and other system background tasks.
>
> `PROCESSWORLD` is intended to be called at the top level of Interlisp, not from within a program. It does not toggle some sort of switch; rather, it constructs some new processes in a new part of the stack, leaving any callers of `PROCESSWORLD` in a now inaccessible part of the stack. Calling `(PROCESSWORLD 'OFF)` is the only way the call to `PROCESSWORLD` ever returns.

`(HARDRESET)` [Function]

> Resets the whole world, and rebuilds the stack from scratch. This is "harder" than doing `RESET` to every process, because it also resets system internal processes (such as the keyboard handler).
>
> `HARDRESET` automatically turns the process world on (or resets it if it was on), unless the variable `AUTOPROCESSFLG` is `NIL`.

Creating and Destroying Processes

(ADD.PROCESS *FORM* $PROP_1$ $VALUE_1$ ... $PROP_N$ $VALUE_N$) [NoSpread Function]

Creates a new process evaluating *FORM*, and returns its process handle. The process's stack environment is the top level, i.e., the new process does not have access to the environment in which ADD.PROCESS was called; all such information must be passed as arguments in *FORM*. The process runs until *FORM* returns or the process is explicitly deleted. An untrapped error within the process also deletes the process (unless its RESTARTABLE property is T), in which case a message is printed to that effect.

The remaining arguments are alternately property names and values. Any property/value pairs acceptable to PROCESSPROP may be given, but the following two are directly relevant to ADD.PROCESS:

NAME — Value should be a litatom; if not given, the process name is taken from (CAR *FORM*). ADD.PROCESS may pack the name with a number to make it unique. This name is solely for the convenience of manipulating processes at Lisp typein; e.g., the name can be given as the *PROC* argument to most process functions, and the name appears in menus of processes. However, programs should normally only deal in process handles, both for efficiency and to avoid the confusion that can result if two processes have the same defining form.

SUSPEND — If the value is non-NIL, the new process is created but then immediately suspended; i.e., the process does not actually run until woken by a WAKE.PROCESS (below).

(PROCESSPROP *PROC PROP NEWVALUE*) [NoSpread Function]

Used to get or set the values of certain properties of process *PROC*, in a manner analogous to WINDOWPROP. If *NEWVALUE* is supplied (including if it is NIL), property *PROP* is given that value. In all cases, returns the old value of the property. The following properties have special meaning for processes; all others are uninterpreted:

NAME — Value is a litatom used for identifying the process to the user.

FORM — Value is the Lisp form used to start the process (readonly).

RESTARTABLE — Value is a flag indicating the disposition of the process following errors or hard resets:

NIL or NO (the default): If an untrapped error (or Control-E or Control-D) causes its form to be exited, the process is deleted. The process is also deleted if a HARDRESET (or Control-D from RAID) occurs, causing the entire Process world to be reinitialized.

T or YES: The process is automatically restarted on errors or HARDRESET. This is the normal setting for persistent "background" processes, such as the mouse process, that can safely restart themselves on errors.

HARDRESET: The process is deleted as usual if an error causes its form to be exited, but it *is* restarted on a HARDRESET. This setting is preferred for persistent processes for which an error is an unusual condition, one that might repeat itself if the process were simply blindly restarted.

RESTARTFORM — If the value is non-NIL, it is the form used if the process is restarted (instead of the value of the FORM property). Of course, the process must also have a non-NIL RESTARTABLE prop for this to have any effect.

BEFOREEXIT — If the value is the atom DON'T, it will not be interrupted by a LOGOUT. If LOGOUT is attempted before the process finishes, a message will appear saying that Interlisp is waiting for the process to finish. If you want the LOGOUT to proceed without waiting, you must use the process status window (from the background menu) to delete the process.

AFTEREXIT — Value indicates the disposition of the process following a resumption of Lisp after some exit (LOGOUT, SYSOUT, MAKESYS). Possible values are:

DELETE: Delete the process.

SUSPEND: Suspend the process; i.e., do not let it run until it is explicitly woken.

An event: Cause the process to be suspended waiting for the event (See the Events section below).

INFOHOOK — Value is a function or form used to provide information about the process, in conjunction with the INFO command in the process status window (see the Process Status Window section below).

WINDOW — Value is a window associated with the process, the process's "main" window. Used to switch the tty process to this process when you click in this window (see the Switching the TTY Process section below).

Setting the WINDOW property does not set the primary I/O stream (NIL) or the terminal I/O stream (T) to the window. When a process is created, I/O operations to the NIL or T stream will cause a new window to appear. TTYDISPLAYSTREAM (see Chapter 28) should be used to set the terminal i/o stream of a process to a specific window.

TTYENTRYFN — Value is a function that is applied to the process when the process is made the tty process (see the Switching the TTY Process section below).

TTYEXITFN — Value is a function that is applied to the process when the process ceases to be the tty process (see the Switching the TTY Process section below).

(THIS.PROCESS) [Function]

Returns the handle of the currently running process, or NIL if the Process world is turned off.

(DEL.PROCESS *PROC* —) [Function]

Deletes process *PROC*. *PROC* may be a process handle (returned by ADD.PROCESS), or its name. If *PROC* is the currently running process, DEL.PROCESS does not return!

(PROCESS.RETURN *VALUE*) [Function]

Terminates the currently running process, causing it to "return" *VALUE*. There is an implicit PROCESS.RETURN around the *FORM* argument given to ADD.PROCESS, so that normally a process can finish by simply returning; PROCESS.RETURN is supplied for earlier termination.

(PROCESS.RESULT *PROCESS WAITFORRESULT*) [Function]

If *PROCESS* has terminated, returns the value, if any, that it returned. This is either the value of a PROCESS.RETURN or the value returned from the form given to ADD.PROCESS. If the process was aborted, the value is NIL. If *WAITFORRESULT* is true, PROCESS.RESULT blocks until *PROCESS* finishes, if necessary; otherwise, it returns NIL immediately if *PROCESS* is still running. *PROCESS* must be the actual process handle returned from

ADD.PROCESS, not a process name, as the association between handle and name disappears when the process finishes (and the process handle itself is then garbage collected if no one else has a pointer to it).

(PROCESS.FINISHEDP *PROCESS*) [Function]

True if *PROCESS* has terminated. The value returned is an indication of how it finished: NORMAL or ERROR.

(PROCESSP *PROC*) [Function]

True if *PROC* is the handle of an active process, i.e., one that has not yet finished.

(RELPROCESSP *PROCHANDLE*) [Function]

True if *PROCHANDLE* is the handle of a deleted process. This is analogous to RELSTKP. It differs from PROCESS.FINISHEDP in that it never causes an error, while PROCESS.FINISHEDP can cause an error if its *PROC* argument is not a process at all.

(RESTART.PROCESS *PROC*) [Function]

Unwinds *PROC* to its top level and reevaluates its form. This is effectively a DEL.PROCESS followed by the original ADD.PROCESS.

(MAP.PROCESSES *MAPFN*) [Function]

Maps over all processes, calling *MAPFN* with three arguments: the process handle, its name, and its form.

(FIND.PROCESS *PROC ERRORFLG*) [Function]

If *PROC* is a process handle or the name of a process, returns the process handle for it, else NIL. If *ERRORFLG* is T, generates an error if *PROC* is not, and does not name, a live process.

Process Control Constructs

(BLOCK *MSECSWAIT TIMER*) [Function]

Yields control to the next waiting process, assuming any is ready to run. If *MSECSWAIT* is specified, it is a number of milliseconds to wait before returning, or T, meaning wait forever (until explicitly woken). Alternatively, *TIMER* can be given as a millisecond timer (as returned by SETUPTIMER, Chapter 12) of an absolute time at which to wake up. In any of those cases, the process enters the *waiting* state until the time limit is up. BLOCK with no arguments leaves the process in the *runnable* state, i.e., it returns as soon as every other runnable process of the same priority has had a chance.

BLOCK can be aborted by interrupts such as Control-D, Control-E, or Control-B. BLOCK will return before its timeout is completed, if the process is woken by WAKE.PROCESS, PROCESS.EVAL, or PROCESS.APPLY.

(DISMISS *MSECSWAIT TIMER NOBLOCK*) [Function]

DISMISS is used to dismiss the current process for a given period of time. Similar to BLOCK, except that:

- DISMISS is guaranteed not to return until the specified time has elapsed
- *MSECSWAIT* cannot be T to wait forever
- If *NOBLOCK* is T, DISMISS will not allow other processes to run, but will busy-wait until the amount of time given has elapsed.

(WAKE.PROCESS *PROC STATUS*) [Function]

Explicitly wakes process *PROC*, i.e., makes it *runnable,* and causes its call to BLOCK (or other waiting function) to return *STATUS*. This is one simple way to notify a process of some happening; however, note that if WAKE.PROCESS is applied to a process more than once before the process actually gets its turn to run, it sees only the latest *STATUS*.

(SUSPEND.PROCESS *PROC*) [Function]

Blocks process *PROC* indefinitely, i.e., *PROC* will not run until it is woken by a WAKE.PROCESS.

The following three functions allow access to the stack context of some other process. They require a little bit of care, and are computationally non-trivial, but they do provide a more powerful way of manipulating another process than WAKE.PROCESS allows.

(PROCESS.EVALV *PROC VAR*) [Function]

Performs (EVALV *VAR*) in the stack context of *PROC*.

(PROCESS.EVAL *PROC FORM WAITFORRESULT*) [Function]

Evaluates *FORM* in the stack context of *PROC*. If *WAITFORRESULT* is true, blocks until the evaluation returns a result, else allows the current process to run in parallel with the evaluation. Any errors that occur will be in the context of *PROC*, so be careful. In particular, note that

(PROCESS.EVAL *PROC* '(NLSETQ (FOO)))

and

```
(NLSETQ (PROCESS.EVAL PROC '(FOO)))
```

behave quite differently if FOO causes an error. And it is quite permissible to intentionally cause an error in proc by performing

```
(PROCESS.EVAL PROC '(ERROR!))
```

If errors are possible and *WAITFORRESULT* is true, the caller should almost certainly make sure that *FORM* traps the errors; otherwise the caller could end up waiting forever if *FORM* unwinds back into the pre-existing stack context of *PROC*.

After *FORM* is evaluated in *PROC*, the process *PROC* is woken up, even if it was running BLOCK or AWAIT.EVENT. This is necessary because an event of interest may have occurred while the process was evaluating *FORM*.

(PROCESS.APPLY *PROC FN ARGS WAITFORRESULT*) [Function]

Performs (APPLY *FN ARGS*) in the stack context of *PROC*. Note the same warnings as with PROCESS.EVAL.

Events

An "event" is a synchronizing primitive used to coordinate related processes, typically producers and consumers. Consumer processes can "wait" on events, and producers "notify" events.

(CREATE.EVENT *NAME*) [Function]

Returns an instance of the EVENT datatype, to be used as the event argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(AWAIT.EVENT *EVENT TIMEOUT TIMERP*) [Function]

Suspends the current process until *EVENT* is notified, or until a timeout occurs. If *TIMEOUT* is NIL, there is no timeout. Otherwise, timeout is either a number of milliseconds to wait, or, if *TIMERP* is T, a millisecond timer set to expire at the desired time using SETUPTIMER (see Chapter 12).

(NOTIFY.EVENT *EVENT ONCEONLY*) [Function]

If there are processes waiting for *EVENT* to occur, causes those processes to be placed in the running state, with *EVENT* returned as the value from AWAIT.EVENT. If *ONCEONLY* is true, only runs the first process waiting for the event (this should only be done if the programmer knows that there can only be one process capable of responding to the event at once).

The meaning of an event is up to the programmer. In general, however, the notification of an event is merely a hint that something of interest to the waiting process has happened; the process should still verify that the conceptual event actually occurred. That is, *the process should be written so that it operates correctly even if woken up before the timeout and in the absence of the notified event.* In particular, the completion of `PROCESS.EVAL` and related operations in effect wakes up the process in which they were performed, since there is no secure way of knowing whether the event of interest occurred while the process was busy performing the `PROCESS.EVAL`.

There is currently one class of system-defined events, used with the network code. Each Pup and NS socket has associated with it an event that is notified when a packet arrives on the socket; the event can be obtained by calling `PUPSOCKETEVENT` or `NSOCKETEVENT`, respectively (see Chapter 32).

Monitors

It is often the case that cooperating processes perform operations on shared structures, and some mechanism is needed to prevent more than one process from altering the structure at the same time. Some languages have a construct called a monitor, a collection of functions that access a common structure with mutual exclusion provided and enforced by the compiler via the use of monitor locks. Interlisp-D has taken this implementation notion as the basis for a mutual exclusion capability suitable for a dynamically-scoped environment.

A monitorlock is an object created by you and associated with (e.g., stored in) some shared structure that is to be protected from simultaneous access. To access the structure, a program waits for the lock to be free, then takes ownership of the lock, accesses the structure, then releases the lock. The functions and macros below are used:

(`CREATE.MONITORLOCK` *NAME* —) [Function]

Returns an instance of the `MONITORLOCK` datatype, to be used as the lock argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(`WITH.MONITOR` *LOCK* $FORM_1$ ... $FORM_N$) [Macro]

Evaluates $FORM_1$ ... $FORM_N$ while owning *LOCK*, and returns the value of $FORM_N$. This construct is implemented so that the lock is released even if the form is exited via error (currently implemented with `RESETLST`).

Ownership of a lock is dynamically scoped: if the current process already owns the lock (e.g., if the caller was itself inside a `WITH.MONITOR` for this lock), `WITH.MONITOR` does not wait for the lock to be free before evaluating $FORM_1$ ... $FORM_N$.

(`WITH.FAST.MONITOR` *LOCK* $FORM_1$ ... $FORM_N$) [Macro]

Like `WITH.MONITOR`, but implemented without the `RESETLST`. User interrupts (e.g., Control-E) are inhibited during the evaluation of $FORM_1$ ... $FORM_N$.

Programming restriction: the evaluation of $FORM_1$ ... $FORM_N$ must not error (the lock would not be released). This construct is mainly useful when the forms perform a small, safe computation that never errors and need never be interrupted.

(`MONITOR.AWAIT.EVENT` *RELEASELOCK EVENT TIMEOUT TIMERP*) [Function]

For use in blocking inside a monitor. Performs (`AWAIT.EVENT` *EVENT TIMEOUT TIMERP*), but releases *RELEASELOCK* first, and reobtains the lock (possibly waiting) on wakeup.

Typical use for `MONITOR.AWAIT.EVENT`: A function wants to perform some operation on *FOO*, but only if it is in a certain state. It has to obtain the lock on the structure to make sure that the state of the structure does not change between the time it tests the state and performs the operation. If the state turns out to be bad, it then waits for some other process to make the state good, meanwhile releasing the lock so that the other process can alter the structure.

```
(WITH.MONITOR FOO-LOCK
  (until CONDITION-OF-FOO
  do (MONITOR.AWAIT.EVENT FOO-LOCK EVENT-FOO-
CHANGED TIMEOUT))
     OPERATE-ON-FOO)
```

It is sometimes convenient for a process to have `WITH.MONITOR` at its top level and then do all its interesting waiting using `MONITOR.AWAIT.EVENT`. Not only is this often cleaner, but in the present implementation in cases where the lock is frequently accessed, it saves the `RESETLST` overhead of `WITH.MONITOR`.

Programming restriction: There must not be an `ERRORSET` between the enclosing `WITH.MONITOR` and the call to `MONITOR.AWAIT.EVENT` such that the `ERRORSET` would catch an `ERROR!` and continue inside the monitor, for the lock would not have been reobtained. (The reason for this restriction is that, although `MONITOR.AWAIT.EVENT` won't itself error, you could have caused an error with an interrupt, or a `PROCESS.EVAL` in the context of the waiting process that produced an error.)

On rare occasions it may be useful to manipulate monitor locks directly. The following two functions are used in the implementation of `WITH.MONITOR`:

(OBTAIN.MONITORLOCK *LOCK DONTWAIT UNWINDSAVE*) [Function]

Takes possession of *LOCK*, waiting if necessary until it is free, unless *DONTWAIT* is true, in which case it returns NIL immediately. If *UNWINDSAVE* is true, performs a RESETSAVE to be unwound when the enclosing RESETLST exits. Returns *LOCK* if *LOCK* was successfully obtained, T if the current process already owned *LOCK*.

(RELEASE.MONITORLOCK *LOCK EVENIFNOTMINE*) [Function]

Releases *LOCK* if it is owned by the current process, and wakes up the next process, if any, waiting to obtain the lock.

If *EVENIFNOTMINE* is non-NIL, the lock is released even if it is not owned by the current process.

When a process is deleted, any locks it owns are released.

Global Resources

The biggest source of problems in the multi-processing environment is the matter of global resources. Two processes cannot both use the same global resource if there can be a process switch in the middle of their use (currently this means calls to BLOCK, but ultimately with a preemptive scheduler means anytime). Thus, user code should be wary of its own use of global variables, if it ever makes sense for the code to be run in more than one process at a time. "State" variables private to a process should generally be bound in that process; structures that are shared among processes (or resources used privately but expensive to duplicate per process) should be protected with monitor locks or some other form of synchronization.

Aside from user code, however, there are many *system* global variables and resources. Most of these arise historically from the single-process Interlisp-10 environment, and will eventually be changed in Interlisp-D to behave appropriately in a multi-processing environment. Some have already been changed, and are described below. Two other resources not generally thought of as global variables—the keyboard and the mouse—are particularly idosyncratic, and are discussed in the next section.

The following resources, which are global in Interlisp-10, are allocated per process in Interlisp-D: primary input and output (the streams affected by **INPUT** and **OUTPUT**), terminal input and output (the streams designated by the name **T**), the primary read table and primary terminal table, and dribble files. Thus, each process can print to its own primary output, print to the terminal, read from a different primary input, all without interfering with another process's reading and printing.

Each process begins life with its primary and terminal input/output streams set to a dummy stream. If the process attempts input or output using any of those dummy streams, e.g., by calling (READ T), or (PRINT & T), a tty window is automatically created for the process, and that window becomes the primary input/output and terminal input/output for the process. The default tty window is created at or near the region specified in the variable DEFAULTTTYREGION.

A process can, of course, call TTYDISPLAYSTREAM explicitly to give itself a tty window of its own choosing, in which case the automatic mechanism never comes into play. Calling TTYDISPLAYSTREAM when a process has no tty window not only sets the terminal streams, but also sets the primary input and output streams to be that window, assuming they were still set to the dummy streams.

(HASTTYWINDOWP *PROCESS*) [Function]

Returns T if the process *PROCESS* has a tty window; NIL otherwise. If *PROCESS* is NIL, it defaults to the current process.

Other system resources that are typically changed by RESETFORM, RESETLST, or RESETVARS are all global entities. In the multiprocessing environment, these constructs are suspect, as there is no provision for "undoing" them when a process switch occurs. For example, in the current release of Interlisp-D, it is not possible to set the print radix to 8 inside only one process, as the print radix is a global entity.

Note that RESETFORM and similar expressions are perfectly valid in the process world, and even quite useful, when they manipulate things strictly within one process. The process world is arranged so that deleting a process also unwinds any RESETxxx expressions that were performed in the process and are still waiting to be unwound, exactly as if a Control-D had reset the process to the top. Additionally, there is an implicit RESETLST at the top of each process, so that RESETSAVE can be used as a way of providing "cleanup" functions for when a process is deleted. For these, the value of RESETSTATE (see Chapter 14) is NIL if the process finished normally, ERROR if it was aborted by an error, RESET if the process was explicitly deleted, and HARDRESET if the process is being restarted after a HARDRESET or a RESTART.PROCESS.

Typein and the TTY Process

There is one global resource, the keyboard, that is particularly problematic to share among processes. Consider, for example, having two processes both performing (READ T). Since the keyboard input routines block while there is no input, both processes would spend most of their time blocking, and it would simply be a matter of chance which process received each character of typein.

To resolve such dilemmas, the system designates a distinguished process, termed the *tty process*, that is assumed to be the process that is involved in terminal interaction. Any typein from the keyboard goes to that process. If a process other than the tty process requests keyboard input, it blocks until it becomes the tty process. When the tty process is switched (in any of the ways described further below), any typeahead that occurred before the switch is saved and associated with the current tty process. Thus, it is always the case that keystrokes are sent to the process that is the tty process at the time of the keystrokes, regardless of when that process actually gets around to reading them.

It is less immediately obvious how to handle keyboard interrupt characters, as their action is asynchronous and not always tied to typein. Interrupt handling is described in the Handling of Interrupts section below.

Switching the TTY Process

Any process can make itself be the tty process by calling `TTY.PROCESS`.

(`TTY.PROCESS` *PROC*) [Function]

Returns the handle of the current tty process. In addition, if *PROC* is non-`NIL`, makes it be the tty process. The special case of *PROC* = `T` is interpreted to mean the executive process; this is sometimes useful when a process wants to explicitly give up being the tty process.

(`TTY.PROCESSP` *PROC*) [Function]

True if *PROC* is the tty process; *PROC* defaults to the running process. Thus, `(TTY.PROCESSP)` is true if the caller is the tty process.

(`WAIT.FOR.TTY` *MSECS NEEDWINDOW*) [Function]

Efficiently waits until `(TTY.PROCESSP)` is true. `WAIT.FOR.TTY` is called internally by the system functions that read from the terminal; user code thus need only call it in special cases.

If *MSECS* is non-`NIL`, it is the number of milliseconds to wait before timing out. If `WAIT.FOR.TTY` times out before `(TTY.PROCESSP)` is true, it returns `NIL`, otherwise it returns `T`. If *MSECS* is `NIL`, `WAIT.FOR.TTY` will not time out.

If *NEEDWINDOW* is non-`NIL`, `WAIT.FOR.TTY` opens a TTY window for the current process if one isn't already open.

`WAIT.FOR.TTY` spawns a new mouse process if called under the mouse process (see `SPAWN.MOUSE`, in the Keeping the Mouse Alive section below).

In some cases, such as in functions invoked as a result of mouse action or a user's typed-in call, it is reasonable for the function to invoke TTY.PROCESS itself so that it can take subsequent user type in. In other cases, however, this is too undisciplined; it is desirable to let the user designate which process typein should be directed to. This is most conveniently done by mouse action.

The system supports the model that "to type to a process, you click in its window." To cooperate with this model, any process desiring keyboard input should put its process handle as the PROCESS property of its window(s). To handle the common case, the function TTYDISPLAYSTREAM does this automatically when the ttydisplaystream is switched to a new window. A process can own any number of windows; clicking in any of those windows gives the process the tty.

This mechanism suffices for most casual process writers. For example, if a process wants all its input/output interaction to occur in a particular window that it has created, it should just make that window be its tty window by calling TTYDISPLAYSTREAM. Thereafter, it can PRINT or READ to/from the T stream; if the process is not the tty process at the time that it calls READ, it will block until the user clicks in the window.

For those needing tighter control over the tty, the default behavior can be overridden or supplemented. The remainder of this section describes the mechanisms involved.

There is a window property WINDOWENTRYFN that controls whether and how to switch the tty to the process owning a window. The mouse handler, before invoking any normal BUTTONEVENTFN, specifically notices the case of a button going down in a window that belongs to a process (i.e., has a PROCESS window property) that is not the tty process. In this case, it invokes the window's WINDOWENTRYFN of one argument (*WINDOW*). WINDOWENTRYFN defaults to GIVE.TTY.PROCESS:

(GIVE.TTY.PROCESS *WINDOW*) [Function]

> If *WINDOW* has a PROCESS property, performs (TTY.PROCESS (WINDOWPROP *WINDOW* 'PROCESS)) and then invokes *WINDOW*'s BUTTONEVENTFN function (or RIGHTBUTTONFN if the right button is down).

There are some cases where clicking in a window does not always imply that the user wants to talk to that window. For example, clicking in a text editor window with a shift key held down means to "shift-select" some piece of text into the input buffer of the *current* tty process. The editor supports this by supplying a WINDOWENTRYFN that performs GIVE.TTY.PROCESS if no shift key is down, but goes into its shift-select mode, without changing the tty process, if a shift key is down. The shift-select mode performs a BKSYSBUF of the selected text when the shift key is let up, the BKSYSBUF feeding input to the current tty process.

Sometimes a process wants to be notified when it becomes the tty process, or stops being the tty process. To support this, there are two process properties, TTYEXITFN and TTYENTRYFN. The actions taken by TTY.PROCESS when it switches the tty to a new process are as follows: the former tty process's TTYEXITFN is called with two arguments (*OLDTTYPROCESS NEWTTYPROCESS*); the new process is made the tty process; finally, the new tty process's TTYENTRYFN is called with two arguments (*NEWTTYPROCESS OLDTTYPROCESS*). Normally the TTYENTRYFN and TTYEXITFN need only their first argument, but the other process involved in the switch is supplied for completeness. In the present system, most processes want to interpret the keyboard in the same way, so it is considered the responsibility of any process that changes the keyboard interpretation to restore it to the normal state by its TTYEXITFN.

A window is "owned" by the last process that anyone gave as the window's PROCESS property. Ordinarily there is no conflict here, as processes tend to own disjoint sets of windows (though, of course, cooperating processes can certainly try to confuse each other). The only likely problem arises with that most global of windows, PROMPTWINDOW. Programs should not be tempted to read from PROMPTWINDOW. This is not usually necessary anyway, as the first attempt to read from T in a process that has not set its TTYDISPLAYSTREAM to its own window causes a tty window to be created for the process (see the Global Resources section above).

Handling of Interrupts

At the time that a keyboard interrupt character (see Chapter 30) is struck, any process could be running, and some decision must be made as to which process to actually interrupt. To the extent that keyboard interrupts are related to typein, most interrupts are taken in the tty process; however, the following are handled specially:

RESET (initially Control-D)
ERROR (initially Control-E) These interrupts are taken in the mouse process, if the mouse is not in its idle state; otherwise they are taken in the tty process. Thus, Control-E can be used to abort some mouse-invoked window action, such as the Shape command. As a consequence, note that if the mouse invokes some lengthy computation that the user thinks of as "background", Control-E still aborts it, even though that may not have been what the user intended. Such lengthy computations, for various reasons, should generally be performed by spawning a separate process to perform them. The RESET interrupt in a process other than the executive is interpreted exactly as if an error unwound the process to its top level: if the process was

designated RESTARTABLE = T, it is restarted; otherwise it is killed.

HELP (initially Control-G)
: A menu of processes is presented to the user, who is asked to select which one the interrupt should occur in. The current tty process appears with a * next to its name at the top of the menu. The menu also includes an entry "[Spawn Mouse]", for the common case of needing a mouse because the mouse process is currently tied up running someone's BUTTONEVENTFN; selecting this entry spawns a new mouse process, and no break occurs.

BREAK (initially Control-B)
: Performs the HELP interrupt in the mouse process, if the mouse is not in its idle state; otherwise it is performed in the tty process.

RUBOUT (initially DELETE)
: This interrupt clears typeahead in *all* processes.

RAID, STACK OVERFLOW
STORAGE FULL
: These interrupts always occur in whatever process was running at the time the interrupt struck. In the cases of STACK OVERFLOW and STORAGE FULL, this means that the interrupt is more likely to strike in the offending process (especially if it is a "runaway" process that is not blocking). Note, however, that this process is still not necessarily the guilty party; it could be an innocent bystander that just happened to use up the last of a resource prodigiously consumed by some other process.

Keeping the Mouse Alive

Since the window mouse handler runs in its own process, it is not available while a window's BUTTONEVENTFN function (or any of the other window functions invoked by mouse action) is running. This leads to two sorts of problems: (1) a long computation underneath a BUTTONEVENTFN deprives the user of the mouse for other purposes, and (2) code that runs as a BUTTONEVENTFN cannot rely on other BUTTONEVENTFNs running, which means that there some pieces of code that run differently from normal when run under the mouse process. These problems are addressed by the following functions:

(SPAWN.MOUSE —) [Function]

Spawns another mouse process, allowing the mouse to run even if it is currently "tied up" under the current mouse process. This function is intended mainly to be typed in at the Lisp executive when the user notices the mouse is busy.

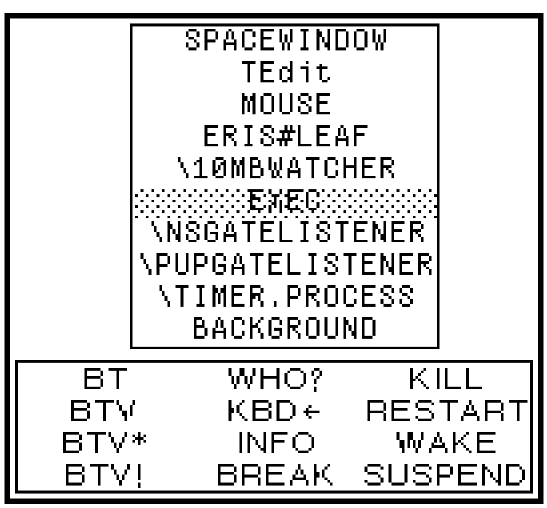
(ALLOW.BUTTON.EVENTS) [Function]

Performs a (SPAWN.MOUSE) only when called underneath the mouse process. This should be called (once, on entry) by any function that relies on BUTTONEVENTFNs for completion, if there is any possibility that the function will itself be invoked by a mouse function.

It never hurts, at least logically, to call SPAWN.MOUSE or ALLOW.BUTTON.EVENTS needlessly, as the mouse process arranges to quietly kill itself if it returns from the user's BUTTONEVENTFN and finds that another mouse process has sprung up in the meantime. (There is, of course, some computational expense.)

Process Status Window

The background menu command PSW (see Chapter 28) and the function PROCESS.STATUS.WINDOW (below) create a "Process Status Window", that allows the user to examine and manipulate all of the existing processes:



The window consists of two menus. The top menu lists all the processes at the moment. Commands in the bottom menu operate on the process selected in the top menu (EXEC in the example above). The commands are:

BT, BTV, BTV*, BTV! Displays a backtrace of the selected process.

WHO? Changes the selection to the tty process, i.e., the one currently in control of the keyboard.

KBD← Associates the keyboard with the selected process; i.e., makes the selected process be the tty process.

INFO If the selected process has an INFOHOOK property, calls it. The hook may be a function, which is then applied to two arguments, the process and the button (LEFT or MIDDLE) used to invoke INFO, or a form, which is simply EVAL'ed. The APPLY or EVAL happens in the context of the selected process, using PROCESS.APPLY or PROCESS.EVAL. The INFOHOOK process property can be set using PROCESSPROP (see the Creating and Destroying Processes section above).

BREAK Enter a break under the selected process. This has the side effect of waking the process with the value returned from the break.

KILL Deletes the selected process.

RESTART Restarts the selected process.

WAKE Wakes the selected process. Prompts for a value to wake it with (see WAKE.PROCESS).

SUSPEND Suspends the selected process; i.e., causes it to block indefinitely (until explicitly woken).

(PROCESS.STATUS.WINDOW *WHERE*) [Function]

Puts up a process status window that provides several debugging commands for manipulating running processes. If the window is already up, PROCESS.STATUS.WINDOW refreshes it. If *WHERE* is a position, the window is placed in that position; otherwise, the user is prompted for a position.

Currently, the process status window runs under the mouse process, like other menus, so if the mouse is unavailable (e.g., a mouse function is performing an extensive computation), you may be unable to use the process status window (you can try SPAWN.MOUSE, of course).

Non-Process Compatibility

This section describes some considerations for authors of programs that ran in the old single-process Interlisp-D environment, and now want to make sure they run properly in the Multi-processing world. The biggest problem to watch out for is code that runs underneath the mouse handler. Writers of mouse handler functions should remember that in the process world the mouse handler runs in its own process, and hence (a) you cannot depend on finding information on the stack (stash it in the window instead), and (b) while your function is running, the mouse is not available (if you have any non-trivial computation to do, spawn a process to do it, notify one of your existing processes to do it, or use PROCESS.EVAL to run it under some other process).

The following functions are meaningful even if the process world is not on: BLOCK (invokes the system background routine, which includes handling the mouse); TTY.PROCESS, THIS.PROCESS (both return NIL); and TTY.PROCESSP (returns T, i.e., anyone is allowed to take tty input). In addition, the following two functions exist in both worlds:

(EVAL.AS.PROCESS *FORM*) [Function]

Same as (ADD.PROCESS *FORM* 'RESTARTABLE 'NO), when processes are running, EVAL when not. This is highly recommended for mouse functions that perform any non-trivial activity.

(EVAL.IN.TTY.PROCESS *FORM WAITFORRESULT*) [Function]

Same as (PROCESS.EVAL (TTY.PROCESS) *FORM WAITFORRESULT*), when processes are running, EVAL when not.

Most of the process functions that do not take a process argument can be called even if processes aren't running. ADD.PROCESS creates, but does not run, a new process (it runs when PROCESSWORLD is called).

23. STREAMS AND FILES

A stream is an object that provides an interface to a physical or logical device. The stream object contains local data and methods that operate on the stream object. Medley's general-purpose I/O functions take a stream as one of their arguments. Not every device is capable of implementing every I/O operation, while some devices offer special functions for that device alone. Such restrictions and extensions are noted in the documentation of each device. The majority of the streams used in Medley fall into two categories: file streams and image streams.

A file is a sequence of data stored on some device that allows the data to be retrieved at a later time. Files are identified by a name specifying their storage devices. Input or output to a file is performed through a stream to the file, using `OPENSTREAM` (below). In addition, there are functions that manipulate the files themselves, rather than their data content.

An image stream is an output stream to a display device, such as the display screen or a printer. In addition to the standard output operations, an image stream implements a variety of graphics operations, such as drawing lines and displaying characters in multiple fonts. Unlike a file, the "content" of an image stream cannot be retrieved. Image streams are described in Chapter 26.

This chapter describes operations specific to file devices: how to name files, how to open streams to files, and how to manipulate files on their devices.

Opening and Closing File Streams

To perform input from or output to a file, you must create a stream to the file, using `OPENSTREAM`:

(**OPENSTREAM** *FILE ACCESS RECOG PARAMETERS* —) [Function]

Opens and returns a stream for the file specified by *FILE*, a file name. *FILE* can be either a string or a symbol. The syntax and manipulation of file names is described at length in the `FILENAMES` section below. Incomplete file names are interpreted with respect to the connected directory (below).

RECOG specifies the recognition mode of *FILE* (below). If *RECOG* = `NIL`, it defaults according to the value of *ACCESS*.

ACCESS specifies the "access rights" to be used when opening the file. Possible values are:

`INPUT` Only input operations are permitted on the already existing file. Starts reading at the beginning of the file. *RECOG* defaults to `OLD`.

`OUTPUT` Only output operations are permitted on the initially empty file. Starts writing at the beginning of the file. While the file is open, other users or processes are unable to open the file for either input or output. *RECOG* defaults to `NEW`.

`BOTH` Both input and output operations are permitted on the file. Starts reading or writing at the beginning of the file. *RECOG* defaults to `OLD/NEW`. *ACCESS* = `BOTH` implies random access (Chapter 25), and may not be possible for files on some devices.

APPEND Only sequential output operations are permitted on the file. Starts writing at the end of the file. *RECOG* defaults to OLD/NEW. *ACCESS* = APPEND may not be allowed for files on some devices.

Note: *ACCESS* = OUTPUT implies that you intend to write a new or different file, even if a version number was specified and the corresponding file already exists. Any previous contents of the file are discarded, and the file is empty immediately after the OPENSTREAM. If you want to write on an already existing file while preserving the old contents, the file must be opened for access BOTH or APPEND.

PARAMETERS is a list of pairs (*ATTRIB VALUE*), where *ATTRIB* is a file attribute (see SETFILEINFO below). A non-list *ATTRIB* in *PARAMETERS* is treated as the pair (*ATTRIB* T). Generally speaking, attributes that belong to the permanent file (e.g., TYPE) can only be set when creating a new file, while attributes that belong only to a particular opening of a file (e.g., ENDOFSTREAMOP) can be set on any call to OPENSTREAM. Not all devices honor all attributes; those not recognized by a particular device are simply ignored.

In addition to the attributes permitted by SETFILEINFO, the following attributes are accepted by OPENSTREAM as values of *ATTRIB* in its *PARAMETERS* argument:

DON'T.CHANGE.DATE If *VALUE* is non-NIL, the file's creation date is not changed when the file is opened. This option is meaningful only for old files opened for BOTH access. You should use this only for specialized applications where the caller does not want the file system to believe the file's content has been changed.

SEQUENTIAL If *VALUE* is non-NIL, this opening of the file need support only sequential access; i.e., the caller intends never to use SETFILEPTR. For some devices, sequential access to files is much more efficient than random access. Note that the device may choose to ignore this attribute and still open the file in a manner that permits random access. Also note that this attribute does not make sense with *ACCESS* = BOTH.

If *FILE* is not recognized by the file system, OPENSTREAM causes the error FILE NOT FOUND. Ordinarily, this error is intercepted via an entry on ERRORTYPELST (Chapter 24), which causes SPELLFILE (see the Searching File Directories below) to be called. SPELLFILE searches alternate directories and possibly attempts spelling correction on the file name. Only if SPELLFILE is unsuccessful will the FILE NOT FOUND error actually occur.

If *FILE* exists but cannot be opened, OPENSTREAM causes one of several other errors: FILE WON'T OPEN if the file is already opened for conflicting access by someone else; PROTECTION VIOLATION if the file is protected against the operation; FILE SYSTEM RESOURCES EXCEEDED if there is no more room in the file system.

(**CLOSEF** *FILE*) [Function]

Closes *FILE* and returns its full file name. Generates an error, FILE NOT OPEN, if *FILE* does not designate an open stream. After closing a stream, no further input/output operations are permitted on it.

If *FILE* is NIL, it is defaulted to the primary input stream if that is not the terminal stream, or else the primary output stream if that is not the terminal stream. If both primary input and output streams are the terminal input/output streams, CLOSEF returns NIL. If CLOSEF closes either the primary input stream or the primary output stream (either explicitly or in the *FILE* = NIL case), it resets the primary stream for that direction to be the corresponding terminal stream. See Chapter 25 for information on the primary input/output streams.

WHENCLOSE (below) allows you to "advise" CLOSEF to perform various operations when a file is closed.

Because of buffering, the contents of a file open for output are not guaranteed to be written to the actual physical file device until CLOSEF is called. Buffered data can be forced out to a file without closing the file by using the function FORCEOUTPUT (Chapter 25).

Some network file devices perform their transactions in the background. As a result, it is possible for a file to be closed by CLOSEF and yet not be "fully" closed for a small time period afterward. During this time the file appears to be busy and cannot be opened for conflicting access by others.

(**CLOSEF?** *FILE*) [Function]

Closes *FILE* if it is open, returning the value of CLOSEF; otherwise does nothing and returns NIL.

In the present implementation of Medley, all open streams to files are kept in a registry of "open files". This registry does not include nameless streams, such as string streams (below), display streams (Chapter 28), and the terminal input and output streams; nor streams explicitly hidden from you, such as dribble streams (Chapter 30). This registry may not persist in future implementations of Medley, but at the present time it is accessible by the following two functions:

(**OPENP** *FILE ACCESS*) [Function]

ACCESS is an access mode for a stream opening (see OPENSTREAM), or NIL for any access.

If *FILE* is a stream, returns its full name if it is open for the specified access, otherwise NIL.

If *FILE* is a file name (a symbol), *FILE* is processed according to the rules of file recognition (below). If a stream open to a file by that name is registered and open for the specified access, then the file's full name is returned. If the file name is not recognized, or no stream is open to the file with the specified access, NIL is returned.

If *FILE* is NIL, returns a list of the full names of all registered streams that are open for the specified access.

(**CLOSEALL** *ALLFLG*) [Function]

Closes all streams in the value of (OPENP). Returns a list of the files closed.

WHENCLOSE (below) allows certain files to be "protected" from CLOSEALL. If *ALLFLG* is T, all files, including those protected by WHENCLOSE, are closed.

File Names

A file name in Medley is a string or symbol whose characters specify a "path" to the actual file: on what host or device the file resides, in which directory, and so forth. Because Medley supports a variety of non-local file devices, parts of the path could be device-dependent. However, it is desirable for programs to be able to manipulate file names in a device-independent manner. To this end, Medley specifies a uniform file name syntax over all devices; the functions that perform the actual file manipulation for a particular device are responsible for any translation to that device's naming conventions.

A file name is composed of a collection of *fields*, some of which have specific meanings. The functions described below refer to each field by a *field name*, a literal atom from among the following: HOST, DEVICE, DIRECTORY, NAME, EXTENSION, and VERSION. The standard syntax for a file name is {HOST}DEVICE:<DIRECTORY>NAME.EXTENSION;VERSION. Some host's file systems do not use all of those fields in their file names.

HOST Specifies the host whose file system contains the file. In the case of local file devices, the "host" is the name of the device, e.g., DSK or FLOPPY.

DEVICE Specifies, for those hosts that divide their file system's name space among mutiple physical devices, the device or logical structure on which the file resides. This should not be confused with Medley's abstract "file device", which denotes either a host or a local physical device and is specified by the HOST field.

DIRECTORY Specifies the "directory" containing the file. A directory usually is a grouping of a possibly large set of loosely related files, e.g., the personal files of a particular user, or the files belonging to some project. The DIRECTORY field usually consists of a principal directory and zero or more subdirectories that together describe a path through a file system's hierarchy. Each subdirectory name is set off from the previous directory or subdirectory by the character ">"; e.g., "LISP>LIBRARY>NEW".

NAME This field carries no specific meaning, but generally names a set of files thought of as being different renditions of the "same" abstract file.

EXTENSION This field also carries no specific meaning, but generally distinguishes the form of files having the same name. Most files systems have some "conventional" extensions that denote something about the content of the file. For example, in Medley, the extension DCOM, LCOM or DFASL denotes files containing compiled function definitions.

VERSION A number used to distinguish the versions or "generations" of the files having a common name and extension. The version number is incremented each time a new file by the same name is created.

Most functions that take as input "a directory" accept either a directory name (the contents of the DIRECTORY field of a file name) or a "full" directory specification—a file name fragment consisting of only the fields HOST, DEVICE, and DIRECTORY. In particular, the "connected directory" (see below) consists, in general, of all three fields.

For convenience in dealing with certain operating systems, Medley also recognizes [] and () as host delimiters (synonymous with {}), and / as a directory delimiter (synonymous with < at the beginning of a directory specification and **>** to terminate directory or subdirectory specification). For example, a file on a Unix file server UNX with the name /usr/foo/bar/stuff.tedit, whose DIRECTORY field is thus usr/foo/bar, could be specified as {UNX}/usr/foo/bar/stuff.tedit, or (UNX)<usr/foo/bar>stuff.tedit, or several other variations. Note that when using [] or () as host delimiters, they usually must be escaped with the reader's escape character if the file name is expressed as a symbol rather than a string.

Different hosts have different requirements for vaild characters in file names. In Medley, all characters are valid. However, in order to be able to parse a file name into its component fields, it is necessary that those characters that are conventionally used as file name delimiters be quoted when they appear inside of fields where there could be ambiguity. The file name quoting character is " ' " (single quote). Thus, the following characters must be quoted when not used as delimeters: >, :, ;, /, and ' itself. The character . (period) need only be quoted if it is to be considered a part of the EXTENSION field. The characters },], and) need only be quoted in a file name when the host field of the name is introduced by {, [, and (, respectively. The characters {, [, (, and < need only be quoted if they appear as the first character of a file name fragment, where they would otherwise be assumed to introduce the HOST or DIRECTORY fields.

The following functions are the standard way to manipulate file names in Medley. Their operation is purely syntactic—they perform no file system operations themselves.

(**UNPACKFILENAME.STRING** *FILENAME*) [Function]

Parses *FILENAME*, returning a list in property list format of alternating field names and field contents. The field contents are returned as strings. If it is a stream, its full name is used.

Only those fields actually present in *FILENAME* are returned. A field is considered present if its delimiting punctuation is present, even if the field itself is empty. Empty fields are denoted by "" (the empty string).

Examples:

```
(UNPACKFILENAME.STRING "FOO.BAR") =>
   (NAME "FOO" EXTENSION "BAR")
(UNPACKFILENAME.STRING "FOO.;2") =>
   (NAME "FOO" EXTENSION "" VERSION "2")
(UNPACKFILENAME.STRING "FOO;") =>
   (NAME "FOO" VERSION "")
(UNPACKFILENAME.STRING
```

```
  "{ERIS}<LISP>CURRENT>IMTRAN.DCOM;21")
=> (HOST "ERIS" DIRECTORY "LISP>CURRENT"
     NAME "IMTRAN" EXTENSION "DCOM"
     VERSION "21")
```

(**UNPACKFILENAME** *FILE*) [Function]

Old version of UNPACKFILENAME.STRING that returns the field values as atoms, rather than as strings. UNPACKFILENAME.STRING is now considered the "correct" way of unpacking file names, because it does not lose information when the contents of a field are numeric. For example,

```
(UNPACKFILENAME 'STUFF.TXT) =>
   (NAME STUFF EXTENSION TXT)
```

but

```
(UNPACKFILENAME 'STUFF.029) =>
   (NAME STUFF EXTENSION 29)
```

Explicitly omitted fields are denoted by the atom NIL, rather than the empty string.

Note: Both UNPACKFILENAME and UNPACKFILENAME.STRING leave the trailing colon on the device field, so that the Tenex device NIL: can be distinguished from the absence of a device. Although UNPACKFILENAME.STRING is capable of making the distinction, it retains this behavior for backward compatibility. Thus,

```
(UNPACKFILENAME.STRING '{TOAST}DSK:FOO) =>
   (HOST "TOAST" DEVICE "DSK:" NAME "FOO")
```

(**FILENAMEFIELD** *FILENAME FIELDNAME*) [Function]

Returns, as an atom, the contents of the *FIELDNAME* field of *FILENAME*. If *FILENAME* is a stream, its full name is used.

(**PACKFILENAME.STRING** *FIELD*$_1$ *CONTENTS*$_1$ ... *FIELD*$_N$ *CONTENTS*$_N$) [NoSpread Function]

Takes a sequence of alternating field names and field contents (atoms or strings), and returns the corresponding file name, as a string.

If PACKFILENAME.STRING is given a single argument, it is interpreted as a list of alternating field names and field contents. Thus PACKFILENAME.STRING and UNPACKFILENAME.STRING operate as inverses.

If the same field name is given twice, the *first* occurrence is used.

The contents of the field name DIRECTORY may be either a directory name or a full directory specification as described above.

PACKFILENAME.STRING also accepts the "field name" BODY to mean that its contents should itself be unpacked and spliced into the argument list at that point. This feature, in conjunction with the rule that fields early in the argument list override later duplicates, is useful for altering existing file names. For example, to provide a default field, place BODY first in the argument list, then the default fields. To override a field, place the new fields first and BODY last.

If the value of the BODY field is a stream, its full name is used.

Examples:

```
(PACKFILENAME.STRING 'DIRECTORY "LISP"
   'NAME "NET")
      =>  "<LISP>NET"

(PACKFILENAME.STRING 'NAME "NET"
   'DIRECTORY "{DSK}<LISPFILES>")
      => "{DSK}<LISPFILES>NET"

(PACKFILENAME.STRING 'DIRECTORY "{DSK}"
   'BODY "{TOAST}<FOO>BAR")
      => "{DSK}BAR"

(PACKFILENAME.STRING 'DIRECTORY "FRED"
   'BODY "{TOAST}<FOO>BAR")
      => "{TOAST}<FRED>BAR"

(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR"
   'DIRECTORY "FRED")
      => "{TOAST}<FOO>BAR"

(PACKFILENAME.STRING 'VERSION NIL
   'BODY "{TOAST}<FOO>BAR.DCOM;2")
      => "{TOAST}<FOO>BAR.DCOM"

(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR.DCOM"
   'VERSION 1)
      => "{TOAST}<FOO>BAR.DCOM;1"

(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR.DCOM;"
   'VERSION 1)
      => "{TOAST}<FOO>BAR.DCOM;"

(PACKFILENAME.STRING 'BODY "BAR.;1"
   'EXTENSION "DCOM")
      => "BAR.;1"

(PACKFILENAME.STRING 'BODY "BAR;1"
   'EXTENSION "DCOM")
      => "BAR.DCOM;1"
```

In the last two examples, note that in one case the extension is explicitly present in the body (as indicated by the preceding period), while in the other there is no indication of an extension, so the default is used.

(**PACKFILENAME** *FIELD*$_1$ *CONTENTS*$_1$ ... *FIELD*$_N$ *CONTENTS*$_N$) [NoSpread Function]

The same as PACKFILENAME.STRING, except that it returns the file name as a symbol, instead of a string.

Incomplete File Names

In general, it is not necessary to pass a complete file name (one containing all the fields listed above) to functions that take a file name as an argument. Interlisp supplies suitable defaults for certain fields (below). Functions that return names of actual files, however, always return the full file name.

If the version field is omitted from a file name, Interlisp performs version recognition, as described below.

If the host, device and/or directory field are omitted from a file name, Interlisp uses the currently connected directory. You can change the currently connected directory by by calling CNDIR (below) or using the programmer's assistant command CONN.

Defaults are added to the partially specified name "left to right" until a host, device or directory field is encountered. Thus, if the connected directory is {TWENTY}PS:<FRED>, then

BAR.DCOM means
{TWENTY}PS:<FRED>BAR.DCOM

<GRANOLA>BAR.DCOM means
{TWENTY}PS:<GRANOLA>BAR.DCOM

MTA0:<GRANOLA>BAR.DCOM means
{TWENTY}MTA0:<GRANOLA>BAR.DCOM

{THIRTY}<GRANOLA>BAR.DCOM means
{THIRTY}<GRANOLA>BAR.DCOM

In addition, if the partially specified name contains a subdirectory, but no principal directory, then the subdirectory is appended to the connected directory. For example,

ISO>BAR.DCOM means
{TWENTY}PS:<FRED>ISO>BAR.DCOM

Or, if the connected directory is the Unix directory {UNX}/usr/fred/, then iso/bar.dcom means {UNX}/usr/fred/iso/bar.dcom, but /other/bar.dcom means {UNX}/other/bar.dcom.

(**CNDIR** *HOST/DIR*) [Function]

Connects to the directory *HOST/DIR*, which can either be a directory name or a full directory specification including host and/or device. If the specification includes just a host, and the host supports directories, the directory is defaulted to the value of (USERNAME); if the host is omitted, connection is made to another directory on the same host as before. If *HOST/DIR* is NIL, connects to the value of LOGINHOST/DIR.

CNDIR returns the full name of the now-connected directory. Causes an error, Non-existent directory, if *HOST/DIR* is not a valid directory.

Note that CNDIR does not necessarily require or provide any directory access privileges. Access privileges are checked when a file is opened.

CONN *HOST/DIR* [Prog. Asst. Command]

Command form of CNDIR for use at the executive. Connects to *HOST/DIR*, or to the value of LOGINHOST/DIR if *HOST/DIR* is omitted. This command is undoable. —Undoing it causes the system to connect to the previously connected directory.

LOGINHOST/DIR [Variable]

CONN with no argument connects to the value of the variable LOGINHOST/DIR, initially {DSK}, but usually reset in your greeting file (Chapter 12).

(**DIRECTORYNAME** *DIRNAME STRPTR*) [Function]

If *DIRNAME* is T, returns the full specification of the currently connected directory. If *DIRNAME* is NIL, returns the value of LOGINHOST/DIR. For any other value of *DIRNAME*, returns a full directory specification if *DIRNAME* designates an existing directory (satisfies DIRECTORYNAMEP), otherwise NIL.

If *STRPTR* is T, the value is returned as an atom, otherwise it is returned as a string.

(**DIRECTORYNAMEP** *DIRNAME HOSTNAME*) [Function]

Returns T if *DIRNAME* is a valid directory on host *HOSTNAME*, or on the host of the currently connected directory if *HOSTNAME* is NIL. *DIRNAME* may be either a directory name or a full directory specification containing host and/or device.

If *DIRNAME* includes subdirectories, this function may or may not pass judgment on their validity. Some hosts support "true" subdirectories, distinct entities manipulable by the file system, while others only provide them as a syntactic convenience.

(**HOSTNAMEP** *NAME*) [Function]

Returns T if *NAME* is recognized as a valid host or file device name at the moment HOSTNAMEP is called.

Version Recognition

Most of the file devices in Interlisp support file version numbers. That is, you can have several files of the exact same name, differing only in their VERSION field, which is incremented for each new "version" of the file that is created. When the filesystem encounters a file name without a version number, it must figure out which version was intended. This process is known as *version recognition*.

When OPENSTREAM opens a file for input and no version number is given, the highest existing version number is used. Similarly, when a file is opened for output and no version number is given, a new file is created with a version number one higher than the highest one currently in use with that file name. You can change he version number defaulting for OPENSTREAM by specifying a different value for its *RECOG* argument (see FULLNAME below).

Other functions that accept file names as arguments generally perform default version recognition, which is newest version for existing files, or a new version if using the file name to create a new file. The one exception is DELFILE, which uses the oldest existing version of the file.

The functions below can be used to perform version recognition without actually calling OPENSTREAM to open the file. Note that these functions only tell the truth at the moment they are called, and thus cannot be used to anticipate the name of the file opened by a comparable OPENSTREAM. They are best used as helpful hints.

(**FULLNAME** *X RECOG*) [Function]

If *X* is an open stream, simply returns the full file name of the stream. Otherwise, if *X* is a file name given as a string or symbol, performs version recognition, as follows:

If *X* is recognized in the recognition mode specified by *RECOG* as an abbreviation for some file, returns the file's full name, otherwise NIL. *RECOG* is one of the following:

OLD — Chooses the newest existing version of the file. Returns NIL if no file named *X* exists.

OLDEST — Chooses the oldest existing version of the file. Returns NIL if no file named *X* exists.

NEW — Chooses a new version of the file. If versions of *X* already exist, then chooses a version number one higher than highest existing version; otherwise chooses version 1. For some file systems, FULLNAME returns NIL if you do not have the access rights necessary to create a new file named *X*.

OLD/NEW — Tries OLD, then NEW. Choose the newest existing version of the file, if any; otherwise chooses version 1. This usually only makes sense if you intend to open *X* for access BOTH.

RECOG = NIL defaults to OLD. For all other values of *RECOG*, generates an error ILLEGAL ARG.

If *X* already contains a version number, the *RECOG* argument will never change it. In particular, *RECOG* = NEW does not require that the file actually be new. For example, (FULLNAME 'FOO.;2 'NEW) may return {ERIS}<LISP>FOO.;2 if that file already exists, even though (FULLNAME 'FOO 'NEW) would default the version to a new number, perhaps returning {ERIS}<LISP>FOO.;5.

(**INFILEP** *FILE*) [Function]

Equivalent to (FULLNAME *FILE* 'OLD). Returns the full file name of the newest version of *FILE* if *FILE* is the name of an existing file that can be opened for input, NIL otherwise.

(**OUTFILEP** *FILE*) [Function]

Equivalent to (FULLNAME *FILE* 'NEW).

Note that INFILEP, OUTFILEP and FULLNAME do not open any files; they are pure predicates. They are also only hints, as they do not imply that the caller has access rights to the file. For example, INFILEP might return non-NIL, but OPENSTREAM might fail for the same file because you don't have read access to it, or the file is open for output by another user. Similarly, OUTFILEP could return non-NIL, but OPENSTREAM could fail with a FILE SYSTEM RESOURCES EXCEEDED error.

Note also that in a shared file system, such as a remote file server, intervening file operations by another user could contradict the information returned by recognition. For example, a file that was INFILEP might be deleted, or between an OUTFILEP and the subsequent OPENSTREAM, another user might create a new version or delete the highest version, causing OPENSTREAM to open a different version of the file than the one returned by OUTFILEP. In addition, some file servers do not support recognition of files in output context. Thus, the "truth" about a file can only be obtained by actually opening the file; creators of files should rely on the name of the stream opened by OPENSTREAM, not the value returned from these recognition functions. In particular, programmers are discouraged from using `OUTFILEP` or `(FULLNAME` *`NAME`* `'NEW)`.

Using File Names Instead of Streams

In earlier implementations of Interlisp, from the days of Interlisp-10 onward, the "handle" used to refer to an open file was not a stream, but rather the file's full name, represented as a symbol. When the file name was passed to any I/O function, it was mapped to a stream by looking it up in a list of open files. This scheme was sometimes convenient for typing in file commands at the executive, but was poor for serious programming in two ways. First, mapping from file name to stream on every input/output operation is inefficient. Second, and more importantly, using the file name as the handle on an open stream means that it is not possible to have more than one stream open on a given file at once.

As of this writing, Medley is in a transition period, where it still supports the use of symbol file names as synonymous with open streams, but this use is not recommended. The remainder of this section discusses this usage of file names for the benefit of those reading older programs and wishing to convert them to work properly when this compatibility feature is removed.

File Name Efficiency Considerations

It is possible for a program to be seriously inefficient using a file name as a stream if the program is not using the name returned by `OPENFILE` (below). Any time that an input/output function is called with a file name other than the full file name, Interlisp must perform recognition on the partial file name to determine which open file is intended. Thus if repeated operations are to be performed, it is considerably more efficient to use the full file name returned from `OPENFILE`.

There is a more subtle problem with partial file names, in that recognition is performed on your entire directory, not just the open files. It is possible for a file name that previously denoted one file to suddenly denote a different file. For example, suppose a program performs `(INFILE 'FOO)`, opening `FOO.;1`, and reads several expressions from `FOO`. Then you interrupt the program, create a `FOO.;2` and resume the program (or a user at another workstation creates a `FOO.;2`). Now a call to `READ` giving it `FOO` as its *`FILE`* argument will generate a `FILE NOT OPEN` error, because `FOO` will be recognized as `FOO.;2`.

Obsolete File Opening Functions

The following functions are now obsolete, but are provided for backwards compatibility:

(**`OPENFILE`** *`FILE ACCESS RECOG PARAMETERS`*) [Function]

Opens *`FILE`* with access rights as specified by *`ACCESS`*, and recognition mode *`RECOG`*, and returns the full name of the resulting stream. Equivalent to `(FULLNAME (OPENSTREAM` *`FILE ACCESS RECOG PARAMETERS`*`))`.

(**`INFILE`** *`FILE`*) [Function]

Opens *`FILE`* for input, and sets it as the primary input stream. Equivalent to `(INPUT (OPENSTREAM` *`FILE`* `'INPUT 'OLD))`

(**OUTFILE** *FILE*) [Function]

Opens *FILE* for output, and sets it as the primary output stream. Equivalent to (OUTPUT (OPENSTREAM *FILE* 'OUTPUT 'NEW)).

(**IOFILE** *FILE*) [Function]

Opens *FILE* for both input and output. Equivalent to (OPENFILE *FILE* 'BOTH 'OLD). Does not affect the primary input or output stream.

Converting Old Programs

At some point in the future, the Medley file system will change so that each call to OPENSTREAM returns a distinct stream, even if a stream is already open to the specified file. This change is required in order to deal with files in a multiprocessing environment.

This change will produce the following incompatibilities:

1. The functions OPENFILE, INPUT, and OUTPUT will return a stream, not a full file name. To make this less confusing in interactive situations, streams will have a print format that reveals the underlying file's actual name.
2. Passing anything other than the object returned from OPENFILE to I/O operations will cause problems. Passing the file's name will be significantly slower than passing the stream (even when passing the "full" file name), and in the case where there is more than one stream open on the file it might even act on the wrong one.
3. OPENP will return NIL when passed the name of a file rather than the value of OPENFILE or OPENSTREAM.

You should consider the following advice when writing new programs and editing existing programs, so your programs will behave properly when the change occurs:

Because of the efficiency and ambiguity considerations described earlier, users have long been encouraged to use only full file names as *FILE* arguments to I/O operations. The "proper" way to have done this was to bind a variable to the value returned from OPENFILE and pass that variable to all I/O operations; such code will continue to work. A less proper way to obtain the full file name, but one which has to date not incurred any obvious penalty, is that which binds a variable to the result of an INFILEP and passes that to OPENFILE and all I/O operations. This has worked because INFILEP and OPENFILE both return a full file name, an invalid assumption in this future world. Such code should be changed to pass around the value of the OPENFILE, not the INFILEP.

Code that calls OPENP to test whether a possibly incomplete file name is already open should be recoded to pass to OPENP only the value returned from OPENFILE or OPENSTREAM.

Code that uses ordinary string functions to manipulate file names, and in particular the value returned from OPENFILE, should be changed to use the the functions UNPACKFILENAME.STRING and PACKFILENAME.STRING. Those functions work both on file names (strings) and streams (coercing the stream to the name of its file).

Code that tests the value of OUTPUT for equality to some known file name or T should be examined carefully and, if possible, recoded.

To see more directly the effects of passing around streams instead of file names, replace your calls to OPENFILE with calls to OPENSTREAM. OPENSTREAM is called in exactly the same way, but returns a STREAM. Streams can be passed to READ, PRINT, CLOSEF, etc just as the file's full name can be currently, but using them is more efficient. The function FULLNAME, when applied to a stream, returns its full file name.

Using Files with Processes

Because Medley does not yet support multiple streams per file, problems can arise if different processes attempt to access the same file. You have to be careful not to have two processes manipulating the same file at the same time, since the two processes will be sharing a single input stream and file pointer. For example, you can't have one process TCOMPL a file while another process is running LISTFILES on it.

File Attributes

Any file has a number of "file attributes", such as the read date, protection, and bytesize. The exact attributes that a file can have is dependent on the file device. The functions GETFILEINFO and SETFILEINFO allow you to access file attributes:

(**GETFILEINFO** *FILE ATTRIB*) [Function]

Returns the current setting of the *ATTRIB* attribute of *FILE*.

(**SETFILEINFO** *FILE ATTRIB VALUE*) [Function]

Sets the attribute *ATTRIB* of *FILE* to be *VALUE*. SETFILEINFO returns T if it is able to change the attribute *ATTRIB*, and NIL if unsuccessful, either because the file device does not recognize *ATTRIB* or because the file device does not permit the attribute to be modified.

The *FILE* argument to GETFILEINFO and SETFILEINFO can be an open stream (or an argument designating an open stream, see Chapter 25), or the name of a closed file. SETFILEINFO in general requires write access to the file.

The attributes recognized by GETFILEINFO and SETFILEINFO fall into two categories: *permanent* attributes, which are properties of the file, and *temporary* attributes, which are properties only of an open stream to the file. The temporary attributes are only recognized when *FILE* designates an open stream; the permanent attributes are usually equally accessible for open and closed files. However, some devices are willing to change the value of certain attributes of an open stream only when specified in the PARAMETERS argument to OPENSTREAM (see above), not on a later call to SETFILEINFO.

The following are permanent attributes of a file:

BYTESIZE The byte size of the file. Medley currently only supports byte size 8.

LENGTH The number of bytes in the file. Alternatively, the byte position of the end-of-file. Like (GETEOFPTR *FILE*), but *FILE* does not have to be open.

SIZE The size of *FILE* in pages.

CREATIONDATE The date and time, as a string, that the content of *FILE* was "created". The creation date changes whenever the content of the file is modified, but remains unchanged when a file is transported, unmodified, across file systems. Specifically, COPYFILE and RENAMEFILE (see below) preserve the file's creation date. Note that this is different from the concept of "creation date" used by some operating systems (e.g., Tops20).

WRITEDATE The date and time, as a string, that the content of *FILE* was last written to this particular file system. When a file is copied, its creation date does not change, but its write date becomes the time at which the copy is made.

READDATE The date and time, as a string, that *FILE* was last read, or NIL if it has never been read.

ICREATIONDATE
IWRITEDATE
IREADDATE The CREATIONDATE, WRITEDATE and READDATE, respectively, in integer form, as IDATE (Chapter 12) would return. This form is useful for comparing dates.

AUTHOR The name of the user who last wrote the file.

TYPE The "type" of the file, some indication of the nature of the file's content. The "types" of files allowed depends on the file device. Most devices recognize the symbol TEXT to mean that the file contains just characters, or BINARY to mean that the file contains arbitrary data.

Some devices support a wider range of file types that distinguish among the various sorts of files one might create whose content is "binary". All devices interpret any value of TYPE that they do not support to be BINARY. Thus, GETFILEINFO may return the more general value BINARY instead of the original type that was passed to SETFILEINFO or OPENSTREAM. Similarly, COPYFILE, while attempting to preserve the TYPE of the file it is copying, may turn, say, an INTERPRESS file into a mere BINARY file.

The way in which some file devices (e.g., Xerox file servers) support a wide range of file types is by representing the type as an integer, whose interpretation is known by the client. The variable FILING.TYPES is used to associate symbolic types with numbers for these devices. This list initially contains some of the well-known assignments of type name to number; you can add additional elements to handle any private file types. For example, suppose there existed an NS file type MAZEFILE with numeric value 5678. You could add the element (MAZEFILE 5678) to FILING.TYPES and then use MAZEFILE as a value for the TYPE attribute to SETFILEINFO or OPENSTREAM. Other devices are, of course, free to store TYPE attributes in whatever manner they wish, be it numeric or symbolic. FILING.TYPES is merely considered the official registry for Xerox file types.

For most file devices, the TYPE of a newly created file, if not specified in the *PARAMETERS* argument to OPENSTREAM, defaults to the value of DEFAULTFILETYPE, initially TEXT.

The following are currently recognized as temporary attributes of an open stream:

ACCESS — The current access rights of the stream (see the beginning of this chapter). Can be one of INPUT, OUTPUT, BOTH, APPEND; or NIL if the stream is not open.

ENDOFSTREAMOP — The action to be taken when a stream is at "end of file" and an attempt is made to take input from it. The value of this attribute is a function of one argument, the stream. The function can examine the stream and its calling context and take any action it wishes. If the function returns normally, its should return either T, meaning to try the input operation again, or the byte that BIN would have returned had there been more bytes to read. Ordinarily, one should not let the ENDOFSTREAMOP function return unless one is only performing binary input from the file, since there is no way in general of knowing in what state the reader was at the time the end of file occurred, and hence how it will interpret a single byte returned to it.

The default ENDOFSTREAMOP is a system function that causes the error END OF FILE. The behavior of that error can be further modified for a particular stream by using the EOF option of WHENCLOSE (see below).

EOL — The end-of-line convention for the stream. This can be CR, LF, or CRLF, indicating with what byte or sequence of bytes the "End Of Line" character is represented on the stream. On input, that sequence of bytes on the stream is read as (CHARCODE EOL) by READCCODE or the string reader. On output, (TERPRI) and (PRINTCCODE (CHARCODE EOL)) cause that sequence of bytes to be placed on the stream.

The end of line convention is usually not apparent to you. The file system is usually aware of the convention used by a particular remote operating system, and sets this attribute accordingly. If you believe a file actually is stored with a different convention than the default, it is possible to modify the default behavior by including the EOL attribute in the *PARAMETERS* argument to OPENSTREAM.

BUFFERS — Value is the number of 512-byte buffers that the stream maintains at one time. This attribute is only used by certain random-access devices (currently, the local disk, floppy, and Leaf servers); all others ignore it.

Streams open to files generally maintain some portion of the file buffered in memory, so that each call to an I/O function does not require accessing the actual file on disk or a file server. For files being read or written sequentially, not much buffer space is needed, since once a byte is read or written, it will never need to be seen again. In the case of random access streams, buffering is more complicated, since a program may jump around in the file, using SETFILEPTR (Chapter 25). In this case, the more buffer space the stream has, the more likely it is that after a SETFILEPTR to a place in the file that has already been accessed, the stream still has that part of the file buffered and need not go out to the device again. This benefit must, of course, be traded off against the amount of memory consumed by the buffers.

NS servers implement the following additional attributes for GETFILEINFO (neither of these attributes are settable with SETFILEINFO):

READER — The name of the user who last read the file.

PROTECTION — A list specifying the access rights to the file. Each element of the list is of the form (name nametype . rights). Name is the name of a user or group or a name pattern. Rights is one or more of the symbols ALL READ WRITE DELETE CREATE or MODIFY. For servers running services 10.0 or later, nametype is the symbol "--". , In earlyer releases it is one of the symbols INDIVIDUAL or GROUP

Closing and Reopening Files

The function WHENCLOSE permits you to associate certain operations with open streams that govern how and when the stream will be closed. You can specify that certain functions will be executed before CLOSEF closes the stream and/or after CLOSEF closes the stream. You can make a particular stream be invisible to CLOSEALL, so that it will remain open across user invocations of CLOSEALL.

(**WHENCLOSE** *FILE* *PROP*$_1$ *VAL*$_1$ . . . *PROP*$_N$ *VAL*$_N$) [NoSpread Function]

FILE must designate an open stream other than T (NIL defaults to the primary input stream, if other than T, or primary output stream if other than T). The remaining arguments specify properties to be associated with the full name of *FILE*. WHENCLOSE returns the full name of *FILE* as its value.

WHENCLOSE recognizes the following property names:

BEFORE — *VAL* is a function that CLOSEF will apply to the stream just before it is closed. This might be used, for example, to copy information about the file from an in-core data structure to the file just before it is closed.

AFTER — *VAL* is a function that CLOSEF will apply to the stream just after it is closed. This capability permits in-core data structures that know about the stream to be cleaned up when the stream is closed.

CLOSEALL — *VAL* is either YES or NO and determines whether *FILE* will be closed by CLOSEALL (YES) or whether CLOSEALL will ignore it (NO). CLOSEALL uses CLOSEF, so that any AFTER functions will be executed if the stream is in fact closed. Files are initialized with CLOSEALL set to YES.

EOF *VAL* is a function that will be applied to the stream when an end-of-file error occurs, and the ERRORTYPELST entry for that error, if any, returns NIL. The function can examine the context of the error, and can decide whether to close the stream, RETFROM some function, or perform some other computation. If the function supplied returns normally (i.e., does not RETFROM some function), the normal error machinery will be invoked.

The default EOF behavior, unless overridden by this WHENCLOSE option, is to call the value of DEFAULTEOFCLOSE (below).

For some applications, the ENDOFSTREAMOP attribute (see above) is a more useful way to intercept the end-of-file error. The ENDOFSTREAMOP attribute comes into effect before the error machinery is ever activated.

Multiple AFTER and BEFORE functions may be associated with a file; they are executed in sequence with the most recently associated function executed first. The CLOSEALL and EOF values, however, will override earlier values, so only the last value specified will have an effect.

DEFAULTEOFCLOSE [Variable]

Value is the name of a function that is called by default when an end of file error occurs and no EOF option has been specified for the stream by WHENCLOSE. The initial value of DEFAULTEOFCLOSE is NILL, meaning take no special action (go ahead and cause the error). Setting it to CLOSEF would cause the stream to be closed before the rest of the error machinery is invoked.

I/O Operations to and from Strings

It is possible to treat a string as if it were the contents of a file by using the following function:

(**OPENSTRINGSTREAM** *STR ACCESS*) [Function]

Returns a stream that can be used to access the characters of the string *STR*. *ACCESS* may be either INPUT, OUTPUT, or BOTH; NIL defaults to INPUT. The stream returned may be used exactly like a file opened with the same access, except that output operations may not extend past the end of the original string. Also, string streams do not appear in the value of (OPENP).

For example, after performing

```
(SETQ STRM (OPENSTRINGSTREAM "THIS 2 (IS A LIST)"))
```

the following succession of reads could occur:

```
(READ STRM)  => THIS
   (RATOM STRM) => 2
   (READ STRM)  => (IS A LIST)
   (EOFP STRM)  => T
```

Compatibility Note: In Interlisp-10 it was possible to take input from a string simply by passing the string as the *FILE* argument to an input function. In order to maintain compatibility with this feature, Medley provides the same capability. This not terribly clean feature persists in the present implementation to give users time to convert old code. This means that strings are *not* equivalent to symbols when specifying a file name as a stream argument. In a future release, the old Interlisp-10 string-reading feature will be decommissioned, and OPENSTRINGSTREAM will be the only way to perform I/O on a string.

Temporary Files and the CORE Device

Many operating systems have a notion of "scratch file", a file typically used as temporary storage for data most naturally maintained in the form of a file, rather than some other data structure. A scratch file can be used as a normal file in most respects, but is automatically deleted from the file system after its useful life is up, e.g., when the job terminates, or you log out. In normal operation, you need never explicitly delete such files, since they are guaranteed to disappear soon.

A similar functionality is provided in Medley by core-resident files. Core-resident files are on the device CORE. The directory structure for this device and all files on it are represented completely within your virtual memory. These files are treated as ordinary files by all file operations; their only distinguishing feature is that all trace of them disappears when the Medley image is abandoned.

Core files are opened and closed by name the same as any other file, e.g., (OPENSTREAM '{CORE}<FOO>FIE.DCOM 'OUTPUT). Directory names are completely optional, so files can also have names of the form {CORE}NAME.EXT. Core files can be enumerated by DIRECTORY (see below). While open, they are registered in (OPENP). They do consume virtual memory space, which is only reclaimed when the file is deleted. Some caution should thus be used when creating large CORE files. Since the virtual memory of an Medley workstation usually persists far longer than the typical process on a mainframe computer, it is still important to delete CORE files after they are no longer in use.

For many applications, the name of the scratch file is irrelevant, and there is no need for anyone to have access to the file independent of the program that created it. For such applications, NODIRCORE files are preferable. Files created on the device lisp NODIRCORE are core-resident files that have no name and are registered in no directory. These files "disappear", and the resources they consume are reclaimed, when all pointers to the file are dropped. Hence, such files need never be explicitly deleted or, for that matter, closed. The "name" of such a file is simply the stream object returned from (OPENSTREAM '{NODIRCORE} 'OUTPUT), and it is this stream object that must be passed to all input/output operations, including CLOSEF and any calls to OPENSTREAM to reopen the file.

(**COREDEVICE** *NAME NODIRFLG*) [Function]

Creates a new device for core-resident files and assigns *NAME* as its device name. Thus, after performing (COREDEVICE 'FOO), one can execute (OPENSTREAM '{FOO}BAR 'OUTPUT) to open a file on that device. Medley is initialized with the single core-resident device named CORE, but COREDEVICE may be used to create any number of logically distinct core devices.

If *NODIRFLG* is non-NIL, a core device that acts like {NODIRCORE} is created.

Compatibility note: In Interlisp-10, it was possible to create scratch files by using file names with suffixes ;S or ;T. In Medley, these suffixes in file names are simply ignored when output is directed to a particular host or device. However, the function PACKFILENAME.STRING is defined to default the device name to CORE if the file has the TEMPORARY attribute and no explicit host is provided.

NULL Device

The NULL device provides a source of content-free "files". (OPENSTREAM '{NULL} 'OUTPUT) creates a stream that discards all output directed at it. (OPENSTREAM '{NULL} 'INPUT**)** creates a stream that is perpetually at end-of-file (i.e., has no input).

Deleting, Copying, and Renaming Files

(**DELFILE** *FILE*) [Function]

Deletes *FILE* if possible. The file must be closed. Returns the full name of the file if deleted, else NIL. Recognition mode for *FILE* is OLDEST, i.e., if *FILE* does not have a version number specified, then DELFILE deletes the oldest version of the file.

(**COPYFILE** *FROMFILE TOFILE*) [Function]

Copies *FROMFILE* to a new file named *TOFILE*. The source and destination may be on any combination of hosts/devices. COPYFILE attempts to preserve the TYPE and CREATIONDATE where possible. If the original file's file type is unknown, COPYFILE attempts to infer the type (file type is BINARY if any of its 8-bit bytes have their high bit on).

COPYFILE uses COPYCHARS (Chapter 25) if the source and destination hosts have different EOL conventions. Thus, it is possible for the source and destination files to be of different lengths.

(**RENAMEFILE** *OLDFILE NEWFILE*) [Function]

Renames *OLDFILE* to be *NEWFILE*. Causes an error, FILE NOT FOUND if *FILE* does not exist. Returns the full name of the new file, if successful, else NIL if the rename cannot be performed.

If *OLDFILE* and *NEWFILE* are on the same host/device, and the device implements a renaming primitive, RENAMEFILE can be very fast. However, if the device does not know how to rename files in place, or if *OLDFILE* and *NEWFILE* are on different devices, RENAMEFILE works by copying *OLDFILE* to *NEWFILE* and then deleting *OLDFILE*.

Searching File Directories

DIRECTORIES [Variable]

Global variable containing the list of directories searched (in order) by SPELLFILE and FINDFILE (below) when not given an explicit *DIRLST* argument. In this list, the atom NIL stands for the login directory (the value of LOGINHOST/DIR), and the atom T stands for the currently connected directory. Other elements should be *full* directory specifications, e.g., {TWENTY}PS:<LISPUSERS>, not merely LISPUSERS.

LISPUSERSDIRECTORIES [Variable]

Global variable containing a list of directories to search for "library" package files. Used by the FILES file package command (Chapter 17).

(**SPELLFILE** *FILE NOPRINTFLG NSFLG DIRLST*) [Function]

Searches for the file name *FILE*, possibly performing spelling correction (see Chapter 20). Returns the corrected file name, if any, otherwise NIL.

If *FILE* has a directory field, SPELLFILE attempts spelling correction against the files in that particular directory. Otherwise, SPELLFILE searches for the file on the directory list *DIRLST* before attempting any spelling correction.

If *NOPRINTFLG* is NIL, SPELLFILE asks you to confirm any spelling correction done, and prints out any files found, even if spelling correction is not done. If *NOPRINTFLG* = T, SPELLFILE does not do any printing, nor ask for approval.

If *NSFLG* = T (or NOSPELLFLG = T, see Chapter 20), no spelling correction is attempted, though searching through *DIRLST* still occurs.

DIRLST is the list of directories searched if *FILE* does not have a directory field. If *DIRLST* is NIL, the value of the variable DIRECTORIES is used.

> Note: If *DIRLST* is NIL, and *FILE* is not found by searching the directories on DIRECTORIES, but the root name of *FILE* has a FILEDATES property (Chapter 17) indicating that a file by that name has been loaded, then the directory indicated in the FILEDATES property is searched, too. This additional search is not done if *DIRLST* is non-NIL.

ERRORTYPELST (Chapter 14) initially contains the entry ((23 (SPELLFILE (CADR ERRORMESS) NIL NOFILESPELLFLG))), which causes SPELLFILE to be called in case of a FILE NOT FOUND error. If the variable NOFILESPELLFLG is T (its initial value), then spelling correction is not done on the file name, but DIRECTORIES is still searched. If SPELLFILE is successful, the operation will be reexecuted with the new (corrected) file name.

(**FINDFILE** *FILE NSFLG DIRLST*) [Function]

Uses SPELLFILE to search for a file named *FILE*. If it finds one, returns its full name, with no user interaction. Specifically, it calls (SPELLFILE *FILE* T *NSFLG DIRLST*), after first performing two simple checks: If *FILE* has an explicit directory, it checks to see if a file so named exists, and if so returns that file. If *DIRLST* is NIL, it looks for *FILE* on the connected directory before calling SPELLFILE.

Listing File Directories

The function DIRECTORY allows you to conveniently specify and/or program a variety of directory operations:

(**DIRECTORY** *FILES COMMANDS DEFAULTEXT DEFAULTVERS*) [Function]

Returns, lists, or performs arbitrary operations on all files specified by the "file group" *FILES*. A file group has the form of a regular file name, except that the character * can be used to match any number of characters, including zero, in the file name. For example, the file group A*B matches all file names beginning with the character A and ending with the character B. The file group *.DCOM matches all files with an extension of DCOM.

If *FILES* does not contain an explicit extension, it is defaulted to *DEFAULTEXT*; if *FILES* does not contain an explicit version, it is defaulted to *DEFAULTVERS*. *DEFAULTEXT* and *DEFAULTVERS* themselves default to *. If the period or semicolon preceding the omitted extension or version, respectively, is present, the field is explicitly empty and no default is used. All other unspecified fields default to *. Null version is interpreted as "highest". Thus *FILES* = * or *.* or *.*;* enumerates all files on the connected directory; *FILES* = *. or *.;* enumerates all versions of files with null extension; *FILES* = *.; enumerates the highest version of files with null extension; and *FILES* = *.*; enumerates the highest version of all files. If *FILES* is NIL, it defaults to *.*;*.

> Note: Some hosts/devices are not capable of supporting "highest version" in enumeration. Such hosts instead enumerate *all* versions.

For each file that matches the file group *FILES*, the "file commands" in *COMMANDS* are executed in order. Some of the file commands allow aborting the command processing for a given file, effectively filtering the list of files. The interpretation of the different file commands is described below. If *COMMANDS* is NIL, it defaults to (COLLECT), which collects the matching file names in a list and returns it as the value of DIRECTORY.

The "file commands" in *COMMANDS* are interpreted as follows:

P — Prints the file's name. For readability, DIRECTORY strips the directory from the name, printing it once as a header in front of each set of consecutive files on the same directory.

PP — Prints the file's name without a version number.

a string — Prints the string.

READDATE, WRITEDATE

CREATIONDATE, SIZE
LENGTH, BYTESIZE
PROTECTION, AUTHOR
TYPE — Prints the appropriate information returned by GETFILEINFO (see above).

COLLECT — Adds the full name of this file to an accumulating list, which will be returned as the value of DIRECTORY.

COUNTSIZE — Adds the size of this file to an accumulating sum, which will be returned as the value of DIRECTORY.

DELETE — Deletes the file.

DELVER — If this file is not the highest version of files by its name, delete it.

PAUSE — Waits until you type any character before proceeding with the rest of the commands (good for display if you want to ponder).

The following commands are predicates to filter the list. If the predicate is not satisfied, then processing for this file is aborted and no further commands (such as those above) are executed for this file.

Note: if the P and PP commands appear in *COMMANDS* ahead of any of the filtering commands below except PROMPT, they are postponed until after the filters. Thus, assuming the caller has placed the attribute options after the filters as well, no printing occurs for a file that is filtered out. This is principally so that functions like DIR (below) can both request printing and pass arbitrary commands through to DIRECTORY, and have the printing happen in the appropriate place.

PROMPT *MESS* — Prompts with the yes / no question *MESS*; if user responds with No, abort command processing for this file.

OLDERTHAN *N* — Continue command processing if the file hasn't been referenced (read or written) in *N* days. *N* can also be a string naming an explicit date and time since which the file must not have been referenced.

NEWERTHAN *N* — Continue command processing if the file has been written within the last *N* days. *N* can also be a string naming an explicit date and time. Note that this is not quite the complement of OLDERTHAN, since it ignores the read date.

BY *USER* — Continue command processing if the file was last written by the given user, i.e., its AUTHOR attribute matches (case insensitively) *USER*.

@ *X* — *X* is either a function of one argument (*FILENAME*), or an arbitrary expression which uses the variable FILENAME freely. If *X* returns NIL, abort command processing for this file.

The following two commands apply not to any particular file, but globally to the manner in which directory information is printed.

OUT *FILE* Directs output to *FILE*.

COLUMNS *N* Attempts to format output in *N* columns (rather than just 1).

DIRECTORY uses the variable DIRCOMMANDS as a spelling list to correct spelling and define abbreviations and synonyms (see Chapter 20). Currently the following abbreviations are recognized:

AU	=>	AUTHOR
-	=>	PAUSE
COLLECT?	=>	PROMPT " ? " COLLECT
DA DATE	=>	CREATIONDATE
TI	=>	WRITEDATE
DEL	=>	DELETE
DEL? DELETE?	=>	PROMPT " delete? " DELETE
OLD	=>	OLDERTHAN 90
PR	=>	PROTECTION
SI	=>	SIZE
VERBOSE	=>	AUTHOR CREATIONDATE SIZE READDATE WRITEDATE

(**FILDIR** *FILEGROUP*) [Function]

Obsolete synonym of (DIRECTORY *FILEGROUP*).

(**DIR** *FILEGROUP* COM_1 ... COM_N) [NLambda NoSpread Function]

Convenient form of DIRECTORY for use in type-in at the executive. Performs (DIRECTORY '*FILEGROUP* '(P COM_1 ... COM_N)).

(**NDIR** *FILEGROUP* COM_1 ... COM_N) [NLambda NoSpread Function]

Version of DIR that lists the file names in a multi-column format. Also, by default only lists the most recent version of files (unless *FILEGROUP* contains an explicit version).

23. STREAMS AND FILES

Medley can perform input/output operations on a large variety of physical devices, including local disk drives, floppy disk drives, the keyboard and display screen, and remote file server computers accessed over a network. While the low-level details of how all these devices perform input/output vary considerably, the Interlisp-D language provides the programmer a small, common set of abstract operations whose use is largely independent of the physical input/output medium involved—operations such as *read, print, change font,* or *go to a new line.* By merely changing the targeted I/O device, a single program can be used to produce output on the display, a file, or a printer.

The underlying data abstraction that permits this flexibility is the *stream*. A stream is a data object (an instance of the data type `STREAM`) that encapsulates all of the information about an input/output connection to a particular I/O device. Each of Medley's general-purpose I/O functions takes a stream as one of its arguments. The general-purpose function then performs action specific to the stream's device to carry out the requested operation. Not every device is capable of implementing every I/O operation, while some devices offer additional functionality by way of special functions for that device alone. Such restrictions and extensions are noted in the documentation of each device.

The vast majority of the streams commonly used in Medley fall into two interesting categories: the *file stream* and the *image stream*.

A file is an ordered collection of data, usually a sequence of characters or bytes, stored on a file device in a manner that allows the data to be retrieved at a later time. Floppy disks, hard disks, and remote file servers are among the devices used to store files. Files are identified by a "file name", which specifies the device on which the file resides and a name unique to a specific file on that device. Input or output to a file is performed by obtaining a stream to the file, using `OPENSTREAM` (see below). In addition, there are functions that manipulate the files themselves, rather than their data content.

An image stream is an output stream to a display device, such as the display screen or a printer. In addition to the standard output operations, such as print, an image stream implements a variety of graphics operations, such as drawing lines and displaying characters in multiple fonts. Unlike a file, the "content" of an image stream cannot be retrieved. Image streams are described in Chapter 26.

The creation of other kinds of streams, such as network byte-stream connections, is described in the chapters peculiar to those kinds of streams. The operations common to streams in general are described in Chapter 24. This chapter describes operations specific to file devices: how to name files, how to open streams to files, and how to manipulate files on their devices.

Opening and Closing File Streams

In order to perform input from or output to a file, it is necessary to create a stream to the file, using `OPENSTREAM`:

(OPENSTREAM *FILE ACCESS RECOG PARAMETERS* —) [Function]

Opens and returns a stream for the file specified by *FILE*, a file name. *FILE* can be either a string or a symbol. The syntax and manipulation of file names is described at length in the FILENAMES section below. Incomplete file names are interpreted with respect to the connected directory (below).

RECOG specifies the recognition mode of *FILE*, as described in a later section of this chapter. If *RECOG* = NIL, it defaults according to the value of *ACCESS*.

ACCESS specifies the "access rights" to be used when opening the file, one of the following:

INPUT Only input operations are permitted on the file. The file must already exist. Starts reading at the beginning of the file. *RECOG* defaults to OLD.

OUTPUT Only output operations are permitted on the file. Starts writing at the beginning of the file, which is initially empty. While the file is open, other users or processes are unable to open the file for either input or output. *RECOG* defaults to NEW.

BOTH Both input and output operations are permitted on the file. Starts reading or writing at the beginning of the file. *RECOG* defaults to OLD/NEW. *ACCESS* = BOTH implies random accessibility (Chapter 25), and thus may not be possible for files on some devices.

APPEND Only sequential output operations are permitted on the file. Starts writing at the *end* of the file. *RECOG* defaults to OLD/NEW. *ACCESS* = APPEND may not be allowed for files on some devices.

Note: *ACCESS* = OUTPUT implies that one intends to write a new or different file, even if a version number was specified and the corresponding file already exists. Thus any previous contents of the file are discarded, and the file is empty immediately after the OPENSTREAM. If it is desired to write on an already existing file while preserving the old contents, the file must be opened for access BOTH or APPEND.

PARAMETERS is a list of pairs (*ATTRIB VALUE*), where *ATTRIB* is any file attribute that the file system is willing to allow you to set (see SETFILEINFO below). A non-list *ATTRIB* in *PARAMETERS* is treated as the pair (*ATTRIB* T). Generally speaking, attributes that belong to the permanent file (e.g., TYPE) can only be set when creating a new file, while attributes that belong only to a particular opening of a file (e.g., ENDOFSTREAMOP) can be set on any call to OPENSTREAM. Not all devices honor all attributes; those not recognized by a particular device are simply ignored.

In addition to the attributes permitted by SETFILEINFO, the following tokens are accepted by OPENSTREAM as values of *ATTRIB* in its *PARAMETERS* argument:

DON'T.CHANGE.DATE — If *VALUE* is non-NIL, the file's creation date is not changed when the file is opened. This option is meaningful only for old files being opened for access BOTH. This should be used only for specialized applications in which the caller does not want the file system to believe the file's content has been changed.

SEQUENTIAL — If *VALUE* is non-NIL, this opening of the file need support only sequential access; i.e., the caller intends never to use SETFILEPTR. For some devices, sequential access to files is much more efficient than random access. Note that the device may choose to ignore this attribute and still open the file in a manner that permits random access. Also note that this attribute does not make sense with *ACCESS* = BOTH.

If *FILE* is not recognized by the file system, OPENSTREAM causes the error FILE NOT FOUND. Ordinarily, this error is intercepted via an entry on ERRORTYPELST (Chapter 24), which causes SPELLFILE (see the Searching File Directories section of this chapter) to be called. SPELLFILE searches alternate directories and possibly attempts spelling correction on the file name. Only if SPELLFILE is unsuccessful will the FILE NOT FOUND error actually occur.

If *FILE* exists but cannot be opened, OPENSTREAM causes one of several other errors: FILE WON'T OPEN if the file is already opened for conflicting access by someone else; PROTECTION VIOLATION if the file is protected against the operation; FILE SYSTEM RESOURCES EXCEEDED if there is no more room in the file system.

(CLOSEF *FILE*) [Function]

Closes *FILE*, and returns its full file name. Generates an error, FILE NOT OPEN, if *FILE* does not designate an open stream. After closing a stream, no further input/output operations are permitted on it.

If *FILE* is NIL, it is defaulted to the primary input stream if that is not the terminal stream, or else the primary output stream if that is not the terminal stream. If both primary input and output streams are the terminal input/output streams, CLOSEF returns NIL. If CLOSEF closes either the primary input stream or the primary output stream (either explicitly or in the *FILE* = NIL case), it resets the primary stream for that direction to be the corresponding terminal stream. See Chapter 25 for information on the primary input/output streams.

WHENCLOSE (see below) allows you to "advise" CLOSEF to perform various operations when a file is closed.

Because of buffering, the contents of a file open for output are not guaranteed to be written to the actual physical file device until CLOSEF is called. Buffered data can be forced out to a file without closing the file by using the function FORCEOUTPUT (Chapter 25).

Some network file devices perform their transactions in the background. As a result, it is possible for a file to be closed by CLOSEF and yet not be "fully" closed for some small period of time afterward, during which time the file appears to still be busy, and cannot be opened for conflicting access by other users.

(CLOSEF? *FILE*) [Function]

Closes *FILE* if it is open, returning the value of CLOSEF; otherwise does nothing and returns NIL.

In the present implementation of Medley, all streams to files are kept, while open, in a registry of "open files". This registry does not include nameless streams, such as string streams (see below), display streams (Chapter 28), and the terminal input and output streams; nor streams explicitly hidden from you, such as dribble streams (Chapter 30). This registry may not persist in future implementations of Medley, but at the present time it is accessible by the following two functions:

(OPENP *FILE ACCESS*) [Function]

ACCESS is an access mode for a stream opening (one of INPUT, OUTPUT, BOTH, or APPEND), or NIL, meaning any access.

If *FILE* is a stream, returns its full name if it is open for the specified access, else NIL.

If *FILE* is a file name (a symbol), *FILE* is processed according to the rules of file recognition (see below). If a stream open to a file by that name is registered and open for the specified access, then the file's full name is returned. If the file name is not recognized, or no stream is open to the file with the specified access, NIL is returned.

If *FILE* is NIL, returns a list of the full names of all registered streams that are open for the specified access.

(CLOSEALL *ALLFLG*) [Function]

Closes all streams in the value of (OPENP). Returns a list of the files closed.

WHENCLOSE (see below) allows certain files to be "protected" from CLOSEALL. If *ALLFLG* is T, all files, including those protected by WHENCLOSE, are closed.

File Names

A file name in Medley is a string or symbol whose characters specify a "path" to the actual file: on what host or device the file resides, in which directory, and so forth. Because Medley supports a variety of non-local file devices, parts of the path could be very device-dependent. However, it is desirable for programs to be able to manipulate file names in a device-independent manner. To this end, Medley specifies a uniform file name syntax over all devices; the functions that perform the actual file manipulation for a particular device are responsible for any translation to that device's naming conventions.

A file name is composed of a collection of *fields*, some of which have specific semantic interpretations. The functions described below refer to each field by a *field name*, a literal atom from among the following: `HOST`, `DEVICE`, `DIRECTORY`, `NAME`, `EXTENSION`, and `VERSION`. The standard syntax for a file name that contains all of those fields is `{HOST}DEVICE:<DIRECTORY>NAME.EXTENSION;VERSION`. Some host's file systems do not use all of those fields in their file names.

`HOST` — Specifies the host whose file system contains the file. In the case of local file devices, the "host" is the name of the device, e.g., `DSK` or `FLOPPY`.

`DEVICE` — Specifies, for those hosts that divide their file system's name space among mutiple physical devices, the device or logical structure on which the file resides. This should not be confused with Medley's abstract "file device", which denotes either a host or a local physical device and is specified by the `HOST` field.

`DIRECTORY` — Specifies the "directory" containing the file. A directory usually is a grouping of a possibly large set of loosely related files, e.g., the personal files of a particular user, or the files belonging to some project. The `DIRECTORY` field usually consists of a principal directory and zero or more subdirectories that together describe a path through a file system's hierarchy. Each subdirectory name is set off from the previous directory or subdirectory by the character ">"; e.g., "`LISP>LIBRARY>NEW`".

`NAME` — This field carries no specific meaning, but generally names a set of files thought of as being different renditions of the "same" abstract file.

`EXTENSION` — This field also carries no specific meaning, but generally distinguishes the form of files having the same name. Most files systems have some "conventional" extensions that denote something about the content of the file. For example, in Medley, the extension `DCOM` standardly denotes a file containing compiled function definitions.

`VERSION` — A number used to distinguish the versions or "generations" of the files having a common name and extension. The version number is incremented each time a new file by the same name is created.

Most functions that take as input "a directory" accept either a directory name (the contents of the `DIRECTORY` field of a file name) or a "full" directory specification—a file name fragment consisting of only the fields HOST, DEVICE, and DIRECTORY. In particular, the "connected directory" (see below) consists, in general, of all three fields.

For convenience in dealing with certain operating systems, Medley also recognizes [] and () as host delimiters (synonymous with {}), and / as a directory delimiter (synonymous with < at the beginning of a directory specification and **>** to terminate directory or subdirectory specification). For example, a file on a Unix file server UNX with the name /usr/foo/bar/stuff.tedit, whose DIRECTORY field is thus usr/foo/bar, could be specified as {UNX}/usr/foo/bar/stuff.tedit, or (UNX)<usr/foo/bar>stuff.tedit, or several other variations. Note that when using [] or () as host delimiters, they usually must be escaped with the reader's % escape character if the file name is expressed as a symbol rather than a string.

Different hosts have different requirements regarding which characters are valid in file names. From Medley's point of view, any characters are valid. However, in order to be able to parse a file name into its component fields, it is necessary that those characters that are conventionally used as file name delimiters be quoted when they appear inside of fields where there could be ambiguity. The file name quoting character is "'" (single quote). Thus, the following characters must be quoted when not used as delimeters: :, >, ;, /, and ' itself. The character . (period) need only be quoted if it is to be considered a part of the EXTENSION field. The characters },], and) need only be quoted in a file name when the host field of the name is introduced by {, [, and (, respectively. The characters {, [, (, and < need only be quoted if they appear as the first character of a file name fragment, where they would otherwise be assumed to introduce the HOST or DIRECTORY fields.

The following functions are the standard way to manipulate file names in Interlisp. Their operation is purely syntactic—they perform no file system operations themselves.

(UNPACKFILENAME.STRING *FILENAME*) [Function]

Parses , returning a list in property list format of alternating field names and field contents. The field contents are returned as strings. If is a stream, its full name is used.

Only those fields actually present in are returned. A field is considered present if its delimiting punctuation (in the case of EXTENSION and VERSION, the preceding period or semicolon, respectively) is present, even if the field itself is empty. Empty fields are denoted by "" (the empty string).

Examples:

```
(UNPACKFILENAME.STRING "FOO.BAR") =>
   (NAME "FOO" EXTENSION "BAR")
(UNPACKFILENAME.STRING "FOO.;2") =>
   (NAME "FOO" EXTENSION "" VERSION "2")
(UNPACKFILENAME.STRING "FOO;") =>
   (NAME "FOO" VERSION "")
(UNPACKFILENAME.STRING
   "{ERIS}<LISP>CURRENT>IMTRAN.DCOM;21")
 => (HOST "ERIS" DIRECTORY "LISP>CURRENT"
      NAME "IMTRAN" EXTENSION "DCOM"
      VERSION "21")
```

(UNPACKFILENAME *FILE*) [Function]

Old version of UNPACKFILENAME.STRING that returns the field values as atoms, rather than as strings. UNPACKFILENAME.STRING is now considered the "correct" way of unpacking file names, because it does not lose information when the contents of a field are numeric. For example,

```
(UNPACKFILENAME 'STUFF.TXT) =>
   (NAME STUFF EXTENSION TXT)
```

but

```
(UNPACKFILENAME 'STUFF.029) =>
   (NAME STUFF EXTENSION 29)
```

Explicitly omitted fields are denoted by the atom NIL, rather than the empty string.

Note: Both UNPACKFILENAME and UNPACKFILENAME.STRING leave the trailing colon on the device field, so that the Tenex device NIL: can be distinguished from the absence of a device. Although UNPACKFILENAME.STRING is capable of making the distinction, it retains this behavior for backward compatibility. Thus,

```
(UNPACKFILENAME.STRING '{TOAST}DSK:FOO) =>
   (HOST "TOAST" DEVICE "DSK:" NAME "FOO")
```

(FILENAMEFIELD *FILENAME FIELDNAME*) [Function]

Returns, as an atom, the contents of the *FIELDNAME* field of *FILENAME*. If *FILENAME* is a stream, its full name is used.

(PACKFILENAME.STRING $FIELD_1$ $CONTENTS_1$ ... $FIELD_N$ $CONTENTS_N$) [NoSpread Function]

Takes a sequence of alternating field names and field contents (atoms or strings), and returns the corresponding file name, as a string.

If PACKFILENAME.STRING is given a single argument, it is interpreted as a list of alternating field names and field contents. Thus PACKFILENAME.STRING and UNPACKFILENAME.STRING operate as inverses.

If the same field name is given twice, the *first* occurrence is used.

The contents of the field name DIRECTORY may be either a directory name or a full directory specification as described above.

PACKFILENAME.STRING also accepts the "field name" BODY to mean that its contents should itself be unpacked and spliced into the argument list at that point. This feature, in conjunction with the rule that fields early in the argument list override later duplicates, is useful for altering existing file names. For example, to provide a default field, place BODY first in the argument list, then the default fields. To override a field, place the new fields first and BODY last.

If the value of the BODY field is a stream, its full name is used.

Examples:

```
(PACKFILENAME.STRING 'DIRECTORY "LISP"
   'NAME "NET")
      =>  "<LISP>NET"

(PACKFILENAME.STRING 'NAME "NET"
   'DIRECTORY "{DSK}<LISPFILES>")
      => "{DSK}<LISPFILES>NET"

(PACKFILENAME.STRING 'DIRECTORY "{DSK}"
   'BODY "{TOAST}<FOO>BAR")
      => "{DSK}BAR"

(PACKFILENAME.STRING 'DIRECTORY "FRED"
   'BODY "{TOAST}<FOO>BAR")
      => "{TOAST}<FRED>BAR"

(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR"
   'DIRECTORY "FRED")
      => "{TOAST}<FOO>BAR"

(PACKFILENAME.STRING 'VERSION NIL
   'BODY "{TOAST}<FOO>BAR.DCOM;2")
      => "{TOAST}<FOO>BAR.DCOM"

(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR.DCOM"
   'VERSION 1)
      => "{TOAST}<FOO>BAR.DCOM;1"

(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR.DCOM;"
   'VERSION 1)
      => "{TOAST}<FOO>BAR.DCOM;"

(PACKFILENAME.STRING 'BODY "BAR.;1"
   'EXTENSION "DCOM")
      => "BAR.;1"

(PACKFILENAME.STRING 'BODY "BAR;1"
   'EXTENSION "DCOM")
      => "BAR.DCOM;1"
```

In the last two examples, note that in one case the extension is explicitly present in the body (as indicated by the preceding period), while in the other there is no indication of an extension, so the default is used.

(PACKFILENAME *FIELD*$_1$ *CONTENTS*$_1$ ... *FIELD*$_N$ *CONTENTS*$_N$) [NoSpread Function]

The same as PACKFILENAME.STRING, except that it returns the file name as a symbol, instead of a string.

Incomplete File Names

In general, it is not necessary to pass a complete file name (one containing all the fields listed above) to functions that take a file name as argument. Interlisp supplies suitable defaults for certain fields, as described below. Functions that return names of actual files, however, always return the fully specified name.

If the version field is omitted from a file name, Interlisp performs version recognition, as described below.

If the host, device and/or directory field are omitted from a file name, Interlisp defaults them with respect to the currently connected directory. The connected directory is changed by calling the function CNDIR or using the programmer's assistant command CONN.

Defaults are added to the partially specified name "left to right" until a host, device or directory field is encountered. Thus, if the connected directory is {TWENTY}PS:<FRED>, then

BAR.DCOM means
  {TWENTY}PS:<FRED>BAR.DCOM

<GRANOLA>BAR.DCOM means
  {TWENTY}PS:<GRANOLA>BAR.DCOM

MTA0:<GRANOLA>BAR.DCOM means
  {TWENTY}MTA0:<GRANOLA>BAR.DCOM

{THIRTY}<GRANOLA>BAR.DCOM means
  {THIRTY}<GRANOLA>BAR.DCOM

In addition, if the partially specified name contains a subdirectory, but no principal directory, then the subdirectory is appended to the connected directory. For example,

ISO>BAR.DCOM means
  {TWENTY}PS:<FRED>ISO>BAR.DCOM

Or, if the connected directory is the Unix directory {UNX}/usr/fred/, then iso/bar.dcom means {UNX}/usr/fred/iso/bar.dcom, but /other/bar.dcom means {UNX}/other/bar.dcom.

(CNDIR *HOST/DIR*) [Function]

Connects to the directory *HOST/DIR*, which can either be a directory name or a full directory specification including host and/or device. If the specification includes just a host, and the host supports directories, the directory is defaulted to the value of (USERNAME); if the host is omitted, connection is made to another directory on the same host as before. If *HOST/DIR* is NIL, connects to the value of LOGINHOST/DIR.

CNDIR returns the full name of the now-connected directory. Causes an error, Non-existent directory, if *HOST/DIR* is not recognized as a valid directory.

Note that CNDIR does not necessarily require or provide any directory access privileges. Access privileges are checked when a file is opened.

CONN *HOST/DIR* [Prog. Asst. Command]

Convenient command form of CNDIR for use at the executive. Connects to *HOST/DIR*, or to the value of LOGINHOST/DIR if *HOST/DIR* is omitted. This command is undoable—undoing it causes the system to connect to the previously connected directory.

LOGINHOST/DIR [Variable]

CONN with no argument connects to the value of the variable LOGINHOST/DIR, initially {DSK}, but usually reset in your greeting file (Chapter 12).

(DIRECTORYNAME *DIRNAME STRPTR*) [Function]

If *DIRNAME* is T, returns the full specification of the currently connected directory. If *DIRNAME* is NIL, returns the "login" directory specification (the value of LOGINHOST/DIR). For any other value of *DIRNAME*, returns a full directory specification if *DIRNAME* designates an existing directory (satisfies DIRECTORYNAMEP), otherwise NIL.

If *STRPTR* is T, the value is returned as an atom, otherwise it is returned as a string.

(DIRECTORYNAMEP *DIRNAME HOSTNAME*) [Function]

Returns T if *DIRNAME* is recognized as a valid directory on host *HOSTNAME*, or on the host of the currently connected directory if *HOSTNAME* is NIL. *DIRNAME* may be either a directory name or a full directory specification containing host and / or device as well.

If *DIRNAME* includes subdirectories, this function may or may not pass judgment on their validity. Some hosts support "true" subdirectories, distinct entities manipulable by the file system, while others only provide them as a syntactic convenience.

(HOSTNAMEP *NAME*) [Function]

Returns T if *NAME* is recognized as a valid host or file device name at the moment HOSTNAMEP is called.

Version Recognition

Most of the file devices in Interlisp support file version numbers. That is, it is possible to have several files of the exact same name, differing only in their VERSION field, which is incremented for each new "version" of the file that is created. When a file name lacking a version number is presented to the file system, it is necessary to determine which version number is intended. This process is known as *version recognition*.

When OPENSTREAM opens a file for input and no version number is given, the highest existing version number is used. Similarly, when a file is opened for output and no version number is given, a new file is created with a version number one higher than the highest one currently in use with that file name. The version number defaulting for OPENSTREAM can be changed by specifying a different value for its *RECOG* argument, as described under FULLNAME, below.

Other functions that accept file names as arguments generally perform the default version recognition, which is newest version for existing files, or a new version if using the file name to create a new file. The one exception is DELFILE, which defaults to the oldest existing version of the file.

The functions below can be used to perform version recognition without actually calling OPENSTREAM to open the file. Note that these functions only tell the truth about the moment at which they are called, and thus cannot in general be used to anticipate the name of the file opened by a comparable OPENSTREAM. They are sometimes, however, helpful hints.

(FULLNAME *X RECOG*) [Function]

If *X* is an open stream, simply returns the full file name of the stream. Otherwise, if *X* is a file name given as a string or symbol, performs version recognition, as follows:

If *X* is recognized in the recognition mode specified by *RECOG* as an abbreviation for some file, returns the file's full name, otherwise NIL. *RECOG* is one of the following:

OLD — Choose the newest existing version of the file. Return NIL if no file named *X* exists.

OLDEST — Choose the oldest existing version of the file. Return NIL if no file named *X* exists.

NEW — Choose a new (not yet existing) version of the file. That is, if versions of *X* already exist, then choose a version number one higher than highest existing version; else choose version 1. For some file systems, FULLNAME returns NIL if you do not have the access rights necessary for creating a new file named *X*.

OLD/NEW — Try OLD, then NEW. That is, choose the newest existing version of the file, if any; else choose version 1. This usually only makes sense if you are intending to open *X* for access BOTH.

RECOG = NIL defaults to OLD. For all other values of *RECOG*, generates an error ILLEGAL ARG.

If *X* already contains a version number, the *RECOG* argument will never change it. In particular, *RECOG* = NEW does not require that the file actually be new. For example, (FULLNAME 'FOO.;2 'NEW) may return {ERIS}<LISP>FOO.;2 if that file already exists, even though (FULLNAME 'FOO 'NEW) would default the version to a new number, perhaps returning {ERIS}<LISP>FOO.;5.

(INFILEP *FILE*) [Function]

Equivalent to (FULLNAME *FILE* 'OLD). That is, returns the full file name of the newest version of *FILE* if *FILE* is recognized as specifying the name of an existing file that could potentially be opened for input, NIL otherwise.

(OUTFILEP *FILE*) [Function]

Equivalent to (FULLNAME *FILE* 'NEW).

Note that INFILEP, OUTFILEP and FULLNAME do not open any files; they are pure predicates. In general they are also only hints, as they do not necessarily imply that the caller has access rights to the file. For example, INFILEP might return non-NIL, but OPENSTREAM might fail for the same file because the file is read-protected against you, or the file happens to be open for output by another user at the time. Similarly, OUTFILEP could return non-NIL, but OPENSTREAM could fail with a FILE SYSTEM RESOURCES EXCEEDED error.

Note also that in a shared file system, such as a remote file server, intervening file operations by another user could contradict the information returned by recognition. For example, a file that was INFILEP might be deleted, or between an OUTFILEP and the subsequent OPENSTREAM, another user might create a new version or delete the highest version, causing OPENSTREAM to open a different version of the file than the one returned by OUTFILEP. In addition, some file servers do not well support recognition of files in output context. Thus, in general, the "truth" about a file can only be obtained by actually opening the file; creators of files should rely on the name of the stream opened by OPENSTREAM, not the value returned from these recognition functions. In particular, for the reasons described earlier, programmers are discouraged from using OUTFILEP or (FULLNAME *NAME* 'NEW).

Using File Names Instead of Streams

In earlier implementations of Interlisp, from the days of Interlisp-10 onward, the "handle" used to refer to an open file was not a stream, but rather the file's full name, represented as a symbol. When the file name was passed to any I/O function, it was mapped to a stream by looking it up in a list of open files. This scheme was sometimes convenient for typing in file commands at the executive, but was very poor for serious programming in two major ways. First, the mapping from file name to stream on every input/output operation is inefficient. Second, and more importantly, using the file name as the handle on an open stream means that it is not possible to have more than one stream open on a given file at once.

As of this writing, Medley is in a transition period, where it still supports the use of symbol file names as synonymous with open streams, but this use is not recommended. The remainder of this section discusses this usage of file names for the benefit of those reading older programs and wishing to convert them as necessary to work properly when this compatibility feature is removed.

File Name Efficiency Considerations

It is possible for a program to be seriously inefficient using a file name as a stream if the program is not using the file's full name, the name returned by OPENFILE (below). Any time that an input/output function is called with a file name other than the full file name, Interlisp must perform recognition on the partial file name in order to determine which open file is intended. Thus if repeated operations are to be performed, it is considerably more efficient to use the full file name returned from `OPENFILE` than to repeatedly use the possibly incomplete name that was used to open the file.

There is a more subtle problem with partial file names, in that recognition is performed on your entire directory, not just the open files. It is possible for a file name that was previously recognized to denote one file to suddenly denote a different file. For example, suppose a program performs `(INFILE 'FOO)`, opening `FOO.;1`, and reads several expressions from `FOO`. Then you interrupt the program, create a `FOO.;2` and resume the program (or a user at another workstation creates a `FOO.;2`). Now a call to `READ` giving it `FOO` as its *FILE* argument will generate a `FILE NOT OPEN` error, because `FOO` will be recognized as `FOO.;2`.

Obsolete File Opening Functions

The following functions are now considered obsolete, but are provided for backwards compatibility:

`(OPENFILE` *FILE ACCESS RECOG PARAMETERS*`)` [Function]

> Opens *FILE* with access rights as specified by *ACCESS*, and recognition mode *RECOG*, and returns the full name of the resulting stream. Equivalent to `(FULLNAME (OPENSTREAM` *FILE ACCESS RECOG PARAMETERS*`))`.

`(INFILE` *FILE*`)` [Function]

> Opens *FILE* for input, and sets it as the primary input stream. Equivalent to `(INPUT (OPENSTREAM` *FILE* `'INPUT 'OLD))`

`(OUTFILE` *FILE*`)` [Function]

> Opens *FILE* for output, and sets it as the primary output stream. Equivalent to `(OUTPUT (OPENSTREAM` *FILE* `'OUTPUT 'NEW))`.

`(IOFILE` *FILE*`)` [Function]

> Equivalent to `(OPENFILE` *FILE* `'BOTH 'OLD)`; opens *FILE* for both input and output. Does not affect the primary input or output stream.

Converting Old Programs

At some point in the future, the Medley file system will change so that each call to `OPENSTREAM` returns a distinct stream, even if a stream is already open to the specified file. This change is required in order to deal rationally with files in a multiprocessing environment.

This change will of necessity produce the following incompatibilities:

1. The functions `OPENFILE`, `INPUT`, and `OUTPUT` will return a `STREAM`, not a full file name. To make this less confusing in interactive situations, `STREAM`s will have a print format that reveals the underlying file's actual name,

2. A greater penalty will ensue for passing as the *FILE* argument to I/O operations anything other than the object returned from OPENFILE. Passing the file's name will be significantly slower than passing the stream (even when passing the "full" file name), and in the case where there is more than one stream open on the file it might even act on the wrong one.

3. OPENP will return NIL when passed the name of a file rather than a stream (the value of OPENFILE or OPENSTREAM).

Users should consider the following advice when writing new programs and editing existing programs, in order that they will continue to operate well when this change is made:

Because of the efficiency and ambiguity considerations described earlier, users have long been encouraged to use only full file names as *FILE* arguments to I/O operations. The "proper" way to have done this was to bind a variable to the value returned from OPENFILE and pass that variable to all I/O operations; such code will continue to work. A less proper way to obtain the full file name, but one which has to date not incurred any obvious penalty, is that which binds a variable to the result of an INFILEP and passes that to OPENFILE and all I/O operations. This has worked because INFILEP and OPENFILE both return a full file name, an invalid assumption in this future world. Such code should be changed to pass around the value of the OPENFILE, not the INFILEP.

Code that calls OPENP to test whether a possibly incomplete file name is already open should be recoded to pass to OPENP only the value returned from OPENFILE or OPENSTREAM.

Code that uses ordinary string functions to manipulate file names, and in particular the value returned from OPENFILE, should be changed to use the the functions UNPACKFILENAME.STRING and PACKFILENAME.STRING. Those functions work both on file names (strings) and streams (coercing the stream to the name of its file).

Code that tests the value of OUTPUT for equality to some known file name or T should be examined carefully and, if possible, recoded.

To see more directly the effects of passing around STREAMs instead of file names, replace your calls to OPENFILE with calls to OPENSTREAM. OPENSTREAM is called in exactly the same way, but returns a STREAM. Streams can be passed to READ, PRINT, CLOSEF, etc just as the file's full name can be currently, but using them is more efficient. The function FULLNAME, when applied to a stream, returns its full file name.

Using Files with Processes

Because Medley does not yet support multiple streams per file, problems can arise if different processes attempt to access the same file. You have to be careful not to have two processes manipulating the same file at the same time, since the two processes will be sharing a single input stream and file pointer. For example, it will not work to have one process TCOMPL a file while another process is running LISTFILES on it.

File Attributes

Any file has a number of "file attributes", such as the read date, protection, and bytesize. The exact attributes that a file can have is dependent on the file device. The functions GETFILEINFO and SETFILEINFO allow you to conveniently access file attributes:

(GETFILEINFO *FILE ATTRIB*) [Function]

Returns the current setting of the *ATTRIB* attribute of *FILE*.

(SETFILEINFO *FILE ATTRIB VALUE*) [Function]

Sets the attribute *ATTRIB* of *FILE* to be *VALUE*. SETFILEINFO returns T if it is able to change the attribute *ATTRIB*, and NIL if unsuccessful, either because the file device does not recognize *ATTRIB* or because the file device does not permit the attribute to be modified.

The *FILE* argument to GETFILEINFO and SETFILEINFO can be an open stream (or an argument designating an open stream, see Chapter 25), or the name of a closed file. SETFILEINFO in general requires write access to the file.

The attributes recognized by GETFILEINFO and SETFILEINFO fall into two categories: *permanent* attributes, which are properties of the file, and *temporary* attributes, which are properties only of an open stream to the file. The temporary attributes are only recognized when *FILE* designates an open stream; the permanent attributes are usually equally accessible for open and closed files. However, some devices are willing to change the value of certain attributes of an open stream only when specified in the *PARAMETERS* argument to OPENSTREAM (see above), not on a later call to SETFILEINFO.

The following are currently recognized as permanent attributes of a file:

BYTESIZE The byte size of the file. Medley currently only supports byte size 8.

LENGTH The number of bytes in the file. Alternatively, the byte position of the end-of-file. Like (GETEOFPTR *FILE*), but *FILE* does not have to be open.

SIZE The size of *FILE* in pages.

CREATIONDATE The date and time, as a string, that the content of *FILE* was "created". The creation date changes whenever the content of the file is modified, but remains unchanged when a file is transported, unmodified, across file systems. Specifically, COPYFILE and RENAMEFILE (see below) preserve the file's creation date. Note that this is different from the concept of "creation date" used by some operating systems (e.g., Tops20).

WRITEDATE The date and time, as a string, that the content of *FILE* was last written to this particular file system. When a file is copied, its creation date does not change, but its write date becomes the time at which the copy is made.

READDATE The date and time, as a string, that *FILE* was last read, or NIL if it has never been read.

ICREATIONDATE
IWRITEDATE
IREADDATE The CREATIONDATE, WRITEDATE and READDATE, respectively, in integer form, as IDATE (Chapter 12) would return. This form is useful for comparing dates.

AUTHOR The name of the user who last wrote the file.

TYPE The "type" of the file, some indication of the nature of the file's content. The "types" of files allowed depends on the file device. Most devices recognize the symbol TEXT to mean that the file contains just characters, or BINARY to mean that the file contains arbitrary data.

Some devices support a wider range of file types that distinguish among the various sorts of files one might create whose content is "binary". All devices interpret any value of TYPE that they do not support to be BINARY. Thus, GETFILEINFO may return the more general value BINARY instead of the original type that was passed to SETFILEINFO or OPENSTREAM. Similarly, COPYFILE, while attempting to preserve the TYPE of the file it is copying, may turn, say, an INTERPRESS file into a mere BINARY file.

The way in which some file devices (e.g., Xerox file servers) support a wide range of file types is by representing the type as an integer, whose interpretation is known by the client. The variable FILING.TYPES is used to associate symbolic types with numbers for these devices. This list initially contains some of the well-known assignments of type name to number; you can add additional elements to handle any private file types. For example, suppose there existed an NS file type MAZEFILE with numeric value 5678. You could add the element (MAZEFILE 5678) to FILING.TYPES and then use MAZEFILE as a value for the TYPE attribute to SETFILEINFO or OPENSTREAM. Other devices are, of course, free to store TYPE attributes in whatever manner they wish, be it numeric or symbolic. FILING.TYPES is merely considered the official registry for Xerox file types.

For most file devices, the TYPE of a newly created file, if not specified in the *PARAMETERS* argument to OPENSTREAM, defaults to the value of DEFAULTFILETYPE, initially TEXT.

The following are currently recognized as temporary attributes of an open stream:

ACCESS — The current access rights of the stream (see the beginning of this chapter). Can be one of INPUT, OUTPUT, BOTH, APPEND; or NIL if the stream is not open.

ENDOFSTREAMOP — The action to be taken when a stream is at "end of file" and an attempt is made to take input from it. The value of this attribute is a function of one argument, the stream. The function can examine the stream and its calling context and take any action it wishes. If the function returns normally, its should return either T, meaning to try the input operation again, or the byte that BIN would have returned had there been more bytes to read. Ordinarily, one should not let the ENDOFSTREAMOP function return unless one is only performing binary input from the file, since there is no way in general of knowing in what state the reader was at the time the end of file occurred, and hence how it will interpret a single byte returned to it.

The default ENDOFSTREAMOP is a system function that causes the error END OF FILE. The behavior of that error can be further modified for a particular stream by using the EOF option of WHENCLOSE (see below).

EOL — The end-of-line convention for the stream. This can be CR, LF, or CRLF, indicating with what byte or sequence of bytes the "End Of Line" character is represented on the stream. On input, that sequence of bytes on the stream is read as (CHARCODE EOL) by READCCODE or the string reader. On output, (TERPRI) and (PRINTCCODE (CHARCODE EOL)) cause that sequence of bytes to be placed on the stream.

The end of line convention is usually not apparent to you. The file system is usually aware of the convention used by a particular remote operating system, and sets this attribute accordingly. If you believe a file actually is stored with a different convention than the default, it is possible to modify the default behavior by including the EOL attribute in the *PARAMETERS* argument to OPENSTREAM.

BUFFERS Value is the number of 512-byte buffers that the stream maintains at one time. This attribute is only used by certain random-access devices (currently, the local disk, floppy, and Leaf servers); all others ignore it.

Streams open to files generally maintain some portion of the file buffered in memory, so that each call to an I/O function does not require accessing the actual file on disk or a file server. For files being read or written sequentially, not much buffer space is needed, since once a byte is read or written, it will never need to be seen again. In the case of random access streams, buffering is more complicated, since a program may jump around in the file, using SETFILEPTR (Chapter 25). In this case, the more buffer space the stream has, the more likely it is that after a SETFILEPTR to a place in the file that has already been accessed, the stream still has that part of the file buffered and need not go out to the device again. This benefit must, of course, be traded off against the amount of memory consumed by the buffers.

Closing and Reopening Files

The function WHENCLOSE permits you to associate certain operations with open streams that govern how and when the stream will be closed. You can specify that certain functions will be executed before CLOSEF closes the stream and/or after CLOSEF closes the stream. You can make a particular stream be invisible to CLOSEALL, so that it will remain open across user invocations of CLOSEALL.

(WHENCLOSE *FILE PROP*$_1$ *VAL*$_1$ . . . *PROP*$_N$ *VAL*$_N$) [NoSpread Function]

FILE must designate an open stream other than T (NIL defaults to the primary input stream, if other than T, or primary output stream if other than T). The remaining arguments specify properties to be associated with the full name of *FILE*. WHENCLOSE returns the full name of *FILE* as its value.

WHENCLOSE recognizes the following property names:

BEFORE — *VAL* is a function that CLOSEF will apply to the stream just before it is closed. This might be used, for example, to copy information about the file from an in-core data structure to the file just before it is closed.

AFTER — *VAL* is a function that CLOSEF will apply to the stream just after it is closed. This capability permits in-core data structures that know about the stream to be cleaned up when the stream is closed.

CLOSEALL — *VAL* is either YES or NO and determines whether *FILE* will be closed by CLOSEALL (YES) or whether CLOSEALL will ignore it (NO). CLOSEALL uses CLOSEF, so that any AFTER functions will be executed if the stream is in fact closed. Files are initialized with CLOSEALL set to YES.

EOF — *VAL* is a function that will be applied to the stream when an end-of-file error occurs, and the ERRORTYPELST entry for that error, if any, returns NIL. The function can examine the context of the error, and can decide whether to close the stream, RETFROM some function, or perform some other computation. If the function supplied returns normally (i.e., does not RETFROM some function), the normal error machinery will be invoked.

The default EOF behavior, unless overridden by this WHENCLOSE option, is to call the value of DEFAULTEOFCLOSE (below).

For some applications, the ENDOFSTREAMOP attribute (see above) is a more useful way to intercept the end-of-file error. The ENDOFSTREAMOP attribute comes into effect before the error machinery is ever activated.

Multiple AFTER and BEFORE functions may be associated with a file; they are executed in sequence with the most recently associated function executed first. The CLOSEALL and EOF values, however, will override earlier values, so only the last value specified will have an effect.

DEFAULTEOFCLOSE [Variable]

Value is the name of a function that is called by default when an end of file error occurs and no EOF option has been specified for the stream by WHENCLOSE. The initial value of DEFAULTEOFCLOSE is NILL, meaning take no special action (go ahead and cause the error). Setting it to CLOSEF would cause the stream to be closed before the rest of the error machinery is invoked.

Local Hard Disk Device

Warning: This section describes the Medley functions that control the local hard disk drive available on some computers. All of these functions may not work on all computers running Medley. For more information on using the local hard disk facilities, see the users guide for your computer.

This section describes the local file system currently supported on the Xerox 1108 and 1186 computers. The Xerox 1132 supports a simpler local file system. The functions below are no-ops on the Xerox 1132, except for DISKPARTITION (which returns a disk partition number), and DISKFREEPAGES. On the Xerox 1132, different numbered partitions are referenced by using devices such as {DSK1}, {DSK2}, etc. {DSK} always refers to the disk partition that Interlisp is running on. The 1132 local file system does not support the use of directories.

The hard disk used with the Xerox 1108 or 1186 may be partitioned into a number of named "logical volumes." Logical volumes may be used to hold the Interlisp virtual memory file (see Chapter 12), or Interlisp files. For information on intializing and partitioning the hard disk, see the users guide for your computer. In order to store Interlisp files on a logical volume, it is necessary to create a lisp file directory on that volume (see CREATEDSKDIRECTORY, below).

So long as there exists a logical volume with a Lisp directory on it, files on this volume can be accessed by using the file device called {DSK}. Medley can be used to read, write, and otherwise interact with files on local disk disks through standard Interlisp input/output functions. All I/O functions such as LOAD, OPENSTREAM, READ, PRINT, GETFILEINFO, COPYFILE, etc., work with files on the local disk.

If you do not have a logical volume with a Lisp directory on it, Interlisp emulates the {DSK} device by a core device, a file device whose backing store is entirely within the Lisp virtual memory. However, this is not recommended because the core device only provides limited scratch space, and since the core device is contained in virtual memory, it (and the files stored on it) will be erased when the virtual memory file is reloaded.

Each logical volume with a Lisp directory on it serves as a directory of the device {DSK}. Files are referred to by forms such as

{DSK}<*VOLUMENAME*>*FILENAME*

Thus, the file INIT.LISP on the volume LISPFILES would be called {DSK}<LISPFILES>INIT.LISP.

Subdirectories within a logical volume are supported, using the > character in file names to delimit subdirectory names. For example, the file name {DSK}<LISPFILES>DOC>DESIGN.TEDIT designates the file names DESIGN.TEDIT on the subdirectory DOC on the logical volume LISPFILES.

If a logical volume name is not specified, it defaults in an unusual but simple way: the logical volume defaults to the next logical volume that has a lisp file directory on it including or after the volume containing the currently running virtual memory. For example, if the local disk has the logical volumes LISP, TEMP, and LISPFILES, the LISP volume contains the running virtual memory, and only the LISP volume has a Lisp file directory on it, then {DSK}INIT.LISP refers to the file {DSK}<LispFiles>INIT.LISP. All the functions below default logical volume names in a similar way, except for those such as CREATEDSKDIRECTORY. To determine the current default lisp file directory, evaluate (DIRECTORYNAME '{DSK}).

(CREATEDSKDIRECTORY *VOLUMENAME*) [Function]

Creates a Lisp file directory on the logical volume *VOLUMENAME*, and returns the name of the directory created. It is only necessary to create a Lisp file directory the first time the logical volume is used. After that, the system automatically recognizes and opens access to the logical volumes that have Lisp file directories on them.

(PURGEDSKDIRECTORY *VOLUMENAME*) [Function]

Erases all Lisp files on the volume *VOLUMENAME*, and deletes the Lisp file directory.

(LISPDIRECTORYP *VOLUMENAME*) [Function]

Returns T if the logical volume *VOLUMENAME* has a lisp file directory on it.

(VOLUMES) [Function]

Returns a list of the names of all of the logical volumes on the local hard disk (whether they have lisp file directories or not).

(VOLUMESIZE *VOLUMENAME*) [Function]

Returns the total size of the logical volume *VOLUMENAME* in disk pages.

(DISKFREEPAGES *VOLUMENAME*) [Function]

Returns the total number of free disk pages left on the logical volume *VOLUMENAME*.

(DISKPARTITION) [Function]

Returns the name of the logical volume containing the virtual memory file that Interlisp is currently running in (see Chapter 12).

(DSKDISPLAY *NEWSTATE*) [Function]

Controls a display window that displays information about the logical volumes on the local hard disk (logical volume names, sizes, free pages, etc.). DSKDISPLAY opens or closes this display window depending on the value of *NEWSTATE* (one of ON, OFF, or CLOSED), and returns the previous state of the display window.

If *NEWSTATE* is ON, the display window is opened, and it is automatically updated whenever the file system state changes (this can slow file operations significantly). If *NEWSTATE* is OFF, the display window is opened, but it is not automatically updated. If *NEWSTATE* is CLOSED, the display window is closed. The display mode is initially set to CLOSED.

Once the display window is open, you can update it or change its state with the mouse. Left-buttoning the display window updates it, and middle-buttoning the window brings up a menu that allows you to change the display state.

Note: DSKDISPLAY uses the value of the variable DSKDISPLAY.POSITION for the position of the lower-left corner of the disk display window when it is opened. This variable is changed if the disk display window is moved.

(SCAVENGEDSKDIRECTORY *VOLUMENAME SILENT*) [Function]

Rebuilds the lisp file directory for the logical volume *VOLUMENAME*. This may repair damage in the unlikely event of file system failure, signified by symptoms such as infinite looping or other strange behavior while the system is doing a directory search. Calling SCAVENGEDSKDIRECTORY will not harm an intact volume.

Normally, SCAVENGEDSKDIRECTORY prints out messages as it scavenges the directory. If *SILENT* is non-NIL, these messages are not printed.

Note: Some low-level disk failures may cause "HARD DISK ERROR" errors to occur. To fix such a failure, it may be necessary to log out of Interlisp, scavenge the logical volume in question using Pilot tools, and then call SCAVENGEDSKDIRECTORY from within Interlisp. See the users guide for your computer for more information.

Floppy Disk Device

Warning: This section describes the Medley functions that control the floppy disk drive available on some computers. All of these functions may not work on all computers running Medley. For more information on using the floppy disk facilities, see the users guide for your computer.

The floppy disk drive is accessed through the device {FLOPPY}. Medley can be used to read, write, and otherwise interact with files on floppy disks through standard Interlisp input/output functions. All I/O functions such as LOAD, OPENSTREAM, READ, PRINT, GETFILEINFO, COPYFILE, etc., work with files on floppies.

Note that floppy disks are a removable storage medium. Therefore, it is only meaningful to perform I/O operations to the floppy disk drive, rather than to a given floppy disk. In this section, the phrase "the floppy" is used to mean "the floppy that is currently in the floppy disk drive."

For example, the following sequence could be used to open a file XXX.TXT on the floppy, print "Hello" on it, and close it:

```
(SETQ XXX (OPENSTREAM '{FLOPPY}XXX.TXT 'OUTPUT 'NEW)
   (PRINT "Hello" XXX)
   (CLOSEF XXX)
```

(FLOPPY.MODE *MODE*) [Function]

Medley can currently read and write files on floppies stored in a number of different formats. At any point, the floppy is considered to be in one of four "modes," which determines how it reads and writes files on the floppy. FLOPPY.MODE sets the floppy mode to the value of *MODE*, one of PILOT, HUGEPILOT, SYSOUT, or CPM, and returns the previous floppy mode. The floppy modes are interpreted as follows:

PILOT — This is the normal floppy mode, using floppies in the Xerox Pilot floppy disk format. This file format allows all of the normal Medley I/O operations. This format also supports file names with arbitrary levels of subdirectories. For example, it is possible to create a file named {FLOPPY}<Lisp>Project>FOO.TXT.

HUGEPILOT — This floppy mode is used to access files that are larger than a single floppy, stored on multiple floppies. There are some restrictions with using "huge" files. Some I/O operations are not meaningful for "huge" files. When a stream is created for output in this mode, the LENGTH file attribute must be specified when the file is opened, so that it is known how many floppies will be needed. When an output file is created, the floppy (or floppies) are automatically erased and reformatted (after confirmation from you).

HUGEPILOT mode is primarily useful for saving big files to and from floppies. For example, the following could be used to copy the file {ERIS}<Lisp>Bigfile.txt onto the huge Pilot file {FLOPPY}BigFile.save:

```
(FLOPPY.MODE 'HUGEPILOT)

(COPYFILE '{ERIS}<Lisp>Bigfile.txt
'{FLOPPY}BigFile.save)
```

and the following would restore the file:

```
(FLOPPY.MODE 'HUGEPILOT)
```

```
(COPYFILE '{FLOPPY}BigFile.save
'{ERIS}<Lisp>Bigfile.txt)
```

During each copying operation, you will be prompted to insert "the next floppy" if {ERIS}<Lisp>Bigfile.txt takes multiple floppies.

SYSOUT Similar to HUGEPILOT mode, SYSOUT mode is used for storing sysout files (Chapter 12) on multiple floppy disks. You are prompted to insert new floppies as they are needed.

This mode is set automatically when SYSOUT or MAKESYS is done to the floppy device: (SYSOUT '{FLOPPY}) or (MAKESYS '{FLOPPY}). Notice that the file name does not need to be specifed in SYSOUT mode; unlike HUGEPILOT mode, the file name Lisp.sysout is always used.

Note: The procedure for loading sysout files from floppies depends on the particular computer being used. For information on loading sysout files from floppies, see the users guide for your computer.

Explicitly setting the mode to SYSOUT is useful when copying a sysout file to or from floppies. For example, the following can be used to copy the sysout file {ERIS}<Lisp>Lisp.sysout onto floppies (it is important to set the floppy mode back when done):

```
(FLOPPY.MODE 'SYSOUT)
(COPYFILE '{ERIS}<Lisp>Lisp.sysout
'{FLOPPY})
(FLOPPY.MODE 'PILOT)
```

CPM Medley supports the single-density single-sided (SDSS) CPM floppy format (a standard used by many computers). CPM-formatted floppies are totally different than Pilot floppies, so you should call FLOPPY.MODE to switch to CPM mode when planning to use CPM floppies. After switching to CPM mode, FLOPPY.FORMAT can be used to create CPM-formatted floppies, and the usual input/output operations work with CPM floppy files.

Note: There are a few limitations on CPM floppy format files: (1) CPM file names are limited to eight or fewer characters, with extensions of three or fewer characters; (2) CPM floppies do not have directories or version numbers; and (3) CPM files are padded out with blanks to make the file lengths multiples of 128.

(FLOPPY.FORMAT *NAME AUTOCONFIRMFLG SLOWFLG*) [Function]

FLOPPY.FORMAT erases and initializes the track information on a floppy disk. This must be done when new floppy disks are to be used for the first time. This can also be used to erase the information on used floppy disks.

NAME should be a string that is used as the name of the floppy (106 characters max). This name can be read and set using FLOPPY.NAME (below).

If *AUTOCONFIRMFLG* is NIL, you will be prompted to confirm erasing the floppy, if it appears to contain valid information. If *AUTOCONFIRMFLG* is T, you are not prompted to confirm.

If *SLOWFLG* is NIL, only the Pilot records needed to give your floppy an empty directory are written. If *SLOWFLG* is T, FLOPPY.FORMAT will completely erase the floppy, writing track information and critical Pilot records on it. *SLOWFLG* should be set to T when formatting a brand-new floppy.

Note: Formatting a floppy is a very compute-intensive operation for the I/O hardware. Therefore, the cursor may stop tracking the mouse and keystrokes may be lost while formatting a floppy. This behavior goes away when the formatting is finished.

Warning: The floppy mode set by FLOPPY.MODE (above) affects how FLOPPY.FORMAT formats the floppy. If the floppy is going to be used in Pilot mode, it should be formatted under (FLOPPY.MODE 'PILOT). If it is to be used as a CMP floppy, it should be formatted under (FLOPPY.MODE 'CPM). The two types of formatting are incompatible.

(FLOPPY.NAME *NAME*) [Function]

If *NAME* is NIL, returns the name stored on the floppy disk. If *NAME* is non-NIL, then the name of the floppy disk is set to *NAME*.

(FLOPPY.FREE.PAGES) [Function]

Returns the number of unallocated free pages on the floppy disk in the floppy disk drive.

Note: Pilot floppy files are represented by contiguous pages on a floppy disk. If you are creating and deleting a lot of files on a floppy, it is advisable to keep such a floppy less than 75 percent full.

(FLOPPY.CAN.READP) [Function]

Returns non-NIL if there is a floppy in the floppy drive.

Note: FLOPPY.CAN.READP does not provide any debouncing (protection against not fully closing the floppy drive door). It may be more useful to use FLOPPY.WAIT.FOR.FLOPPY (below).

(FLOPPY.CAN.WRITEP) [Function]

Returns non-NIL if there is a floppy in the floppy drive and the floppy drive can write on this floppy.

It is not possible to write on a floppy disk if the "write-protect notch" on the floppy disk is punched out.

(FLOPPY.WAIT.FOR.FLOPPY *NEWFLG*) [Function]

If *NEWFLG* is NIL, waits until a floppy is in the floppy drive before returning.

If *NEWFLG* is T, waits until the existing floppy in the floppy drive, if any, is removed, then waits for a floppy to be inserted into the drive before returning.

(FLOPPY.SCAVENGE) [Function]

Attempts to repair a floppy whose critical records have become confused (causing errors when file operations are attempted). May also retrieve accidently-deleted files, provided they haven't been overwritten by new files.

(FLOPPY.TO.FILE *TOFILE*) [Function]

Copies the entire contents of the floppy to the "floppy image" file *TOFILE*, which can be on a file server, local disk, etc. This can be used to create a centralized copy of a floppy, that different users can copy to their own floppy disks (using FLOPPY.FROM.FILE).

Note: A floppy image file for an 8-inch floppy is about 2500 pages long, regardless of the number of pages in use on the floppy.

(FLOPPY.FROM.FILE *FROMFILE*) [Function]

Copies the "floppy image" file *FROMFILE* to the floppy. *FROMFILE* must be a file produced by FLOPPY.TO.FILE.

(FLOPPY.ARCHIVE *FILES NAME*) [Function]

FLOPPY.ARCHIVE formats a floppy inserted into the floppy drive, giving the floppy the name *NAME*#1. FLOPPY.ARCHIVE then copies each file in *FILES* to the freshly formatted floppy. If the first floppy fills up, FLOPPY.ARCHIVE uses multiple floppies (named *NAME*#2, *NAME*#3, etc.), each time prompting you to insert a new floppy.

The function DIRECTORY (see below) is convenient for generating a list of files to archive. For example,

```
(FLOPPY.ARCHIVE
   (DIRECTORY '{ERIS}<Lisp>Project>*)
   'Project)
```

will archive all files on the directory {ERIS}<Lisp>Project> to floppies (named Project#1, Project#2, etc.).

(FLOPPY.UNARCHIVE *HOST/DIRECTORY*) [Function]

FLOPPY.UNARCHIVE copies all files on the current floppy to the directory *HOST/DIRECTORY*. For example, (FLOPPY.UNARCHIVE '{ERIS}<Lisp>Project>) will copy each file on the current floppy to the directory {ERIS}<Lisp>Project>. If there is more than one floppy to restore from archive, FLOPPY.UNARCHIVE should be called on each floppy disk.

I/O Operations to and from Strings

It is possible to treat a string as if it were the contents of a file by using the following function:

(OPENSTRINGSTREAM *STR ACCESS*) [Function]

Returns a stream that can be used to access the characters of the string *STR*. *ACCESS* may be either INPUT, OUTPUT, or BOTH; NIL defaults to INPUT. The stream returned may be used exactly like a file opened with the same access, except that output operations may not extend past the end of the original string. Also, string streams do not appear in the value of (OPENP).

For example, after performing

```
(SETQ STRM (OPENSTRINGSTREAM "THIS 2 (IS A LIST)"))
```

the following succession of reads could occur:

```
(READ STRM)  => THIS
   (RATOM STRM) => 2
   (READ STRM)  => (IS A LIST)
   (EOFP STRM)  => T
```

Compatibility Note: In Interlisp-10 it was possible to take input from a string simply by passing the string as the *FILE* argument to an input function. In order to maintain compatibility with this feature, Medley provides the same capability. This not terribly clean feature persists in the present implementation to give users time to convert old code. This means that strings are *not* equivalent to symbols when specifying a file name as a stream argument. In a future release, the old Interlisp-10 string-reading feature will be decommissioned, and OPENSTRINGSTREAM will be the only way to perform I/O on a string.

Temporary Files and the CORE Device

Many operating systems have a notion of "scratch file", a file typically used as temporary storage for data most naturally maintained in the form of a file, rather than some other data structure. A scratch file can be used as a normal file in most respects, but is automatically deleted from the file system after its useful life is up, e.g., when the job terminates, or you log out. In normal operation, you need never explicitly delete such files, since they are guaranteed to disappear soon.

A similar functionality is provided in Medley by core-resident files. Core-resident files are on the device CORE. The directory structure for this device and all files on it are represented completely within your virtual memory. These files are treated as ordinary files by all file operations; their only distinguishing feature is that all trace of them disappears when the virtual memory is abandoned.

Core files are opened and closed by name the same as any other file, e.g., (OPENSTREAM '{CORE}<FOO>FIE.DCOM 'OUTPUT). Directory names are completely optional, so files can also have names of the form {CORE}NAME.EXT. Core files can be enumerated by DIRECTORY (see below). While open, they are registered in (OPENP). They do consume virtual memory space, which is only reclaimed when the file is deleted. Some caution should thus be used when creating large CORE files. Since the virtual memory of an Medley workstation usually persists far longer than the typical process on a mainframe computer, it is still important to delete CORE files after they are no longer in use.

For many applications, the name of the scratch file is irrelevant, and there is no need for anyone to have access to the file independent of the program that created it. For such applications, NODIRCORE files are preferable. Files created on the device lisp NODIRCORE are core-resident files that have no name and are registered in no directory. These files "disappear", and the resources they consume are reclaimed, when all pointers to the file are dropped. Hence, such files need never be explicitly deleted or, for that matter, closed. The "name" of such a file is simply the stream object returned from (OPENSTREAM '{NODIRCORE} 'OUTPUT), and it is this stream object that must be passed to all input/output operations, including CLOSEF and any calls to OPENSTREAM to reopen the file.

(COREDEVICE *NAME NODIRFLG*) [Function]

Creates a new device for core-resident files and assigns *NAME* as its device name. Thus, after performing (COREDEVICE 'FOO), one can execute (OPENSTREAM '{FOO}BAR 'OUTPUT) to open a file on that device. Medley is initialized with the single core-resident device named CORE, but COREDEVICE may be used to create any number of logically distinct core devices.

If *NODIRFLG* is non-NIL, a core device that acts like {NODIRCORE} is created.

Compatibility note: In Interlisp-10, it was possible to create scratch files by using file names with suffixes ;S or ;T. In Medley, these suffixes in file names are simply ignored when output is directed to a particular host or device. However, the function PACKFILENAME.STRING is defined to default the device name to CORE if the file has the TEMPORARY attribute and no explicit host is provided.

NULL Device

The NULL device provides a source of content-free "files". (OPENSTREAM '{NULL} 'OUTPUT) creates a stream that discards all output directed at it. (OPENSTREAM '{NULL} 'INPUT**)** creates a stream that is perpetually at end-of-file (i.e., has no input).

Deleting, Copying, and Renaming Files

(DELFILE *FILE*) [Function]

Deletes *FILE* if possible. The file must be closed. Returns the full name of the file if deleted, else NIL. Recognition mode for *FILE* is OLDEST, i.e., if *FILE* does not have a version number specified, then DELFILE deletes the oldest version of the file.

(COPYFILE *FROMFILE TOFILE*) [Function]

Copies *FROMFILE* to a new file named *TOFILE*. The source and destination may be on any combination of hosts/devices. COPYFILE attempts to preserve the TYPE and CREATIONDATE where possible. If the original file's file type is unknown, COPYFILE attempts to infer the type (file type is BINARY if any of its 8-bit bytes have their high bit on).

COPYFILE uses COPYCHARS (Chapter 25) if the source and destination hosts have different EOL conventions. Thus, it is possible for the source and destination files to be of different lengths.

(RENAMEFILE *OLDFILE NEWFILE*) [Function]

Renames *OLDFILE* to be *NEWFILE*. Causes an error, FILE NOT FOUND if *FILE* does not exist. Returns the full name of the new file, if successful, else NIL if the rename cannot be performed.

If *OLDFILE* and *NEWFILE* are on the same host/device, and the device implements a renaming primitive, RENAMEFILE can be very fast. However, if the device does not know how to rename files in place, or if *OLDFILE* and *NEWFILE* are on different devices, RENAMEFILE works by copying *OLDFILE* to *NEWFILE* and then deleting *OLDFILE*.

Searching File Directories

DIRECTORIES [Variable]

Global variable containing the list of directories searched (in order) by SPELLFILE and FINDFILE (below) when not given an explicit *DIRLST* argument. In this list, the atom NIL stands for the login directory (the value of LOGINHOST/DIR), and the atom T stands for the currently connected directory. Other elements should be *full* directory specifications, e.g., {TWENTY}PS:<LISPUSERS>, not merely LISPUSERS.

`LISPUSERSDIRECTORIES` [Variable]

Global variable containing a list of directories to search for "library" package files. Used by the `FILES` file package command (Chapter 17).

`(SPELLFILE` *FILE NOPRINTFLG NSFLG DIRLST*`)` [Function]

Searches for the file name *FILE*, possibly performing spelling correction (see Chapter 20). Returns the corrected file name, if any, otherwise `NIL`.

If *FILE* has a directory field, `SPELLFILE` attempts spelling correction against the files in that particular directory. Otherwise, `SPELLFILE` searches for the file on the directory list *DIRLST* before attempting any spelling correction.

If *NOPRINTFLG* is `NIL`, `SPELLFILE` asks you to confirm any spelling correction done, and prints out any files found, even if spelling correction is not done. If *NOPRINTFLG* = `T`, `SPELLFILE` does not do any printing, nor ask for approval.

If *NSFLG* = `T` (or `NOSPELLFLG` = `T`, see Chapter 20), no spelling correction is attempted, though searching through *DIRLST* still occurs.

DIRLST is the list of directories searched if *FILE* does not have a directory field. If *DIRLST* is `NIL`, the value of the variable `DIRECTORIES` is used.

> Note: If *DIRLST* is `NIL`, and *FILE* is not found by searching the directories on `DIRECTORIES`, but the root name of *FILE* has a `FILEDATES` property (Chapter 17) indicating that a file by that name has been loaded, then the directory indicated in the `FILEDATES` property is searched, too. This additional search is not done if *DIRLST* is non-`NIL`.

`ERRORTYPELST` (Chapter 14) initially contains the entry `((23 (SPELLFILE (CADR ERRORMESS) NIL NOFILESPELLFLG)))`, which causes `SPELLFILE` to be called in case of a `FILE NOT FOUND` error. If the variable `NOFILESPELLFLG` is `T` (its initial value), then spelling correction is not done on the file name, but `DIRECTORIES` is still searched. If `SPELLFILE` is successful, the operation will be reexecuted with the new (corrected) file name.

`(FINDFILE` *FILE NSFLG DIRLST*`)` [Function]

Uses `SPELLFILE` to search for a file named *FILE*. If it finds one, returns its full name, with no user interaction. Specifically, it calls `(SPELLFILE` *FILE* `T` *NSFLG DIRLST*`)`, after first performing two simple checks: If *FILE* has an explicit directory, it checks to see if a file so named exists, and if so returns that file. If *DIRLST* is `NIL`, it looks for *FILE* on the connected directory before calling `SPELLFILE`.

Listing File Directories

The function DIRECTORY allows you to conveniently specify and/or program a variety of directory operations:

(DIRECTORY *FILES COMMANDS DEFAULTEXT DEFAULTVERS*) [Function]

Returns, lists, or performs arbitrary operations on all files specified by the "file group" *FILES*. A file group has the form of a regular file name, except that the character * can be used to match any number of characters, including zero, in the file name. For example, the file group A*B matches all file names beginning with the character A and ending with the character B. The file group *.DCOM matches all files with an extension of DCOM.

If *FILES* does not contain an explicit extension, it is defaulted to *DEFAULTEXT*; if *FILES* does not contain an explicit version, it is defaulted to *DEFAULTVERS*. *DEFAULTEXT* and *DEFAULTVERS* themselves default to *. If the period or semicolon preceding the omitted extension or version, respectively, is present, the field is explicitly empty and no default is used. All other unspecified fields default to *. Null version is interpreted as "highest". Thus *FILES* = * or *.* or *.*;* enumerates all files on the connected directory; *FILES* = *. or *.;* enumerates all versions of files with null extension; *FILES* = *.; enumerates the highest version of files with null extension; and *FILES* = *.*; enumerates the highest version of all files. If *FILES* is NIL, it defaults to *.*;*.

> Note: Some hosts/devices are not capable of supporting "highest version" in enumeration. Such hosts instead enumerate *all* versions.

For each file that matches the file group *FILES*, the "file commands" in *COMMANDS* are executed in order. Some of the file commands allow aborting the command processing for a given file, effectively filtering the list of files. The interpretation of the different file commands is described below. If *COMMANDS* is NIL, it defaults to (COLLECT), which collects the matching file names in a list and returns it as the value of DIRECTORY.

The "file commands" in *COMMANDS* are interpreted as follows:

P — Prints the file's name. For readability, DIRECTORY strips the directory from the name, printing it once as a header in front of each set of consecutive files on the same directory.

PP — Prints the file's name without a version number.

a string — Prints the string.

READDATE, WRITEDATE
CREATIONDATE, SIZE
LENGTH, BYTESIZE
PROTECTION, AUTHOR
TYPE — Prints the appropriate information returned by GETFILEINFO (see above).

`COLLECT` Adds the full name of this file to an accumulating list, which will be returned as the value of `DIRECTORY`.

`COUNTSIZE` Adds the size of this file to an accumulating sum, which will be returned as the value of `DIRECTORY`.

`DELETE` Deletes the file.

`DELVER` If this file is not the highest version of files by its name, delete it.

`PAUSE` Waits until you type any character before proceeding with the rest of the commands (good for display if you want to ponder).

The following commands are predicates to filter the list. If the predicate is not satisfied, then processing for this file is aborted and no further commands (such as those above) are executed for this file.

Note: if the `P` and `PP` commands appear in *COMMANDS* ahead of any of the filtering commands below except `PROMPT`, they are postponed until after the filters. Thus, assuming the caller has placed the attribute options after the filters as well, no printing occurs for a file that is filtered out. This is principally so that functions like `DIR` (below) can both request printing and pass arbitrary commands through to `DIRECTORY`, and have the printing happen in the appropriate place.

`PROMPT` *MESS* Prompts with the yes/no question *MESS*; if user responds with `No`, abort command processing for this file.

`OLDERTHAN` *N* Continue command processing if the file hasn't been referenced (read or written) in *N* days. *N* can also be a string naming an explicit date and time since which the file must not have been referenced.

`NEWERTHAN` *N* Continue command processing if the file has been written within the last *N* days. *N* can also be a string naming an explicit date and time. Note that this is not quite the complement of `OLDERTHAN`, since it ignores the read date.

`BY` *USER* Continue command processing if the file was last written by the given user, i.e., its `AUTHOR` attribute matches (case insensitively) *USER*.

`@` *X* *X* is either a function of one argument (*FILENAME*), or an arbitrary expression which uses the variable `FILENAME` freely. If *X* returns `NIL`, abort command processing for this file.

The following two commands apply not to any particular file, but globally to the manner in which directory information is printed.

`OUT` *`FILE`*	Directs output to *`FILE`*.
`COLUMNS` *`N`*	Attempts to format output in *`N`* columns (rather than just 1).

`DIRECTORY` uses the variable `DIRCOMMANDS` as a spelling list to correct spelling and define abbreviations and synonyms (see Chapter 20). Currently the following abbreviations are recognized:

```
      AU  =>  AUTHOR
       -  =>  PAUSE
COLLECT?  =>  PROMPT " ? " COLLECT
      DA
    DATE  =>  CREATIONDATE
      TI  =>  WRITEDATE
     DEL  =>  DELETE
    DEL?
 DELETE?  =>  PROMPT " delete? " DELETE
     OLD  =>  OLDERTHAN 90
      PR  =>  PROTECTION
      SI  =>  SIZE
 VERBOSE  =>  AUTHOR CREATIONDATE SIZE
          READDATE WRITEDATE
```

`(FILDIR` *`FILEGROUP`*`)` [Function]

Obsolete synonym of `(DIRECTORY` *`FILEGROUP`*`)`.

`(DIR` *`FILEGROUP`* COM_1 `...` COM_N`)` [NLambda NoSpread Function]

Convenient form of `DIRECTORY` for use in type-in at the executive. Performs `(DIRECTORY '`*`FILEGROUP`*` '(P` COM_1 `...` COM_N`))`.

`(NDIR` *`FILEGROUP`* COM_1 `...` COM_N`)` [NLambda NoSpread Function]

Version of `DIR` that lists the file names in a multi-column format. Also, by default only lists the most recent version of files (unless *`FILEGROUP`* contains an explicit version).

File Servers

A file server is a shared resource on a local communications network which provides large amounts of file storage. Different file servers honor a variety of access protocols. Medley supports the following protocols: PUP-FTP, PUP-Leaf, and NS Filing. In addition, there are library packages available that support other communications protocols, such as TCP/IP and RS232.

With the exception of the RS232-based protocols, which exist only for file transfer, these network protocols are integrated into the Medley file system to allow files on a file server to be treated in much the same way files are accessed on local devices, such as the disk. Thus, it is possible to call OPENSTREAM on the file {ERIS}<LISP>FOO.DCOM;3 and read from it or write to it just as if the file had been on the local disk ({DSK}<LISP>FOO.DCOM;3), rather than on a remote server named ERIS. However, the protocols vary in how much control they give the workstation over file system operations. Hence, some restrictions apply, as described in the following sections.

PUP File Server Protocols

There are two file server protocols in the family of PUP protocols: Leaf and FTP. Some servers support both, while others support only one of them. Medley uses whichever protocol is more appropriate for the requested operation.

Leaf is a random access protocol, so files opened using these protocols are RANDACCESSP, and thus most normal I/O operations can be performed. However, Leaf does not support directory enumeration. Hence, DIRECTORY cannot be used on a Leaf file server unless the server also supports FTP. In addition, Leaf does not supply easy access to a file's attributes. INFILEP and GETFILEINFO have to open the file for input in order to obtain their information, and hence the file's read date will change, even though the semantics of these functions do not imply it.

FTP is a file transfer protocol that only permits sequential access to files. However, most implementations of it are considerably more efficient than Leaf. Medley uses FTP in preference to Leaf whenever the call to OPENSTREAM requests sequential access only. In particular, the functions SYSOUT and COPYFILE open their files for sequential access. If a file server supports FTP but for some reason it is undesirable for Lisp to use it, one can set the internal variable \FTPAVAILABLE to NIL.

The system normally maintains a Leaf connection to a host in the background. This connection can be broken by calling (BREAKCONNECTION *HOST*). Any subsequent reference to files on that host will re-establish the connection. The principal use for this function arises when you interrupt a file operation in such a way that the file server thinks the file is open but Lisp thinks it is closed (or not yet open). As a result, the next time Lisp tries to open the file, it gets a file busy error.

Xerox NS File Server Protocols

Interlisp supports file access to Xerox 803x file servers, using the Filing Protocol built on Xerox Network Systems protocols. Medley determines that a host is an NS File Server by the presense of a colon in its name, e.g., {PHYLEX:}. The general format of NS fileserver device names is {*SERVERNAME:DOMAIN:ORGANIZATION*}; the device specification for an 8000-series product in general includes the ClearingHouse domain and organization. If domain and organization are not supplied directly, then they are obtained from the defaults, which themselves are found by consulting the nearest ClearingHouse if you have not defined them in an init file. However, note that the server name must still have a colon in it to distinguish it from other types of host names (e.g., PUP server names).

NS file servers in general permit arbitrary characters in file names. You should be cognizant of file name quoting conventions, and the fact that any file name presented as a symbol needs to have characters of significance to the reader, such as space, escaped with a %. Of course, one can always present the file name as a string, in which case only the quoting conventions are important.

NS file servers support a true hierarchical file system, where subdirectories are just another kind of file, which needs to be explicitly created. In Interlisp, subdirectories are created automatically as needed: A call to `OPENFILE` to create a file in a non-existent subdirectory automatically creates the subdirectory. `CONN` to a non-existent subdirectory asks you whether to create the directory. For those using Star software, a directory corresponds to a "File Drawer," while a subdirectory corresponds to a "File Folder."

Because of their hierarchical structure, NS directories can be enumeerated to arbitrary levels. The default is to enumerate all the files (the leaves of the tree), omitting the subdirectory nodes themselves. This default can be changed by the following variable:

`FILING.ENUMERATION.DEPTH` [Variable]

This variable is either a number, specifying the number of levels deep to enumeerate, or `T`, meaning enumerate to all levels. In the former case, when the enumeration reaches the specified depth, only the subdirectory name rooted at that level is listed, and none of its descendants is listed. When `FILING.ENUMERATION.DEPTH` is `T`, all files are listed, and no subdirectory names are listed. `FILING.ENUMERATION.DEPTH` is initially `T`.

Independent of `FILING.ENUMERATION.DEPTH`, a request to enumerate the top-level of a file server’s hierarchy lists only the top level, i.e., assumes a depth of 1. For example, `(DIRECTORY '{PHYLEX:})` lists exactly the top-level directories of the server `PHYLEX:`.

NS file servers do not currently support random access. Therefore, `SETFILEPTR` of an NS file generally causes an error. However, `GETFILEPTR` returns the correct character position for open files on NS file servers. In addition, `SETFILEPTR` works in the special case where the file is open for input, and the file pointer is being set forward. In this case, the intervening charactgers are automatically read.

Even while Interlisp has no file open on an NS Server, the system maintains a "session" with the server for a while in order to improve the speed of subsequent requests to the server. While this session is open, it is possible for some nodes of the server’s file system to appear "busy" or inaccessible to certain clients on other workstations (such as Star). If this happens, the following function can be used to terminate any open sessions immediately.

`(BREAK.NSFILING.CONNECTION` *`HOST`*`)` [Function]

Closes any open connections to NS file server *`HOST`*.

Operating System Designations

Some of the network server protocols are implemented on more than one kind of foreign host. Such hosts vary in their conventions for logging in, naming files, representing end-of-line, etc. In order for Interlisp to communicate gracefully with all these hosts, it is necessary that the variable NETWORKOSTYPES be set correctly.The following functions are now considered obsolete, but are provided for backwards compatibility:

NETWORKOSTYPES [Variable]

An association-list that associates a host name with its operating system type. Elements in this list are of the form (*HOSTNAME* . *TYPE*). For example, (MAXC2 . TENEX). The operating system types currently known to Lisp are TENEX, TOPS20, UNIX, and VMS. The host names in this list should be the "canonical" host name, represented as an uppercase atom. For PUP and NS hosts, the function CANONICAL.HOSTNAME (below) can be used to determine which of several aliases of a server is the canonical name.

(CANONICAL.HOSTNAME *HOSTNAME*) [Function]

Returns the "canonical" name of the server *HOSTNAME*, or NIL if *HOSTNAME* is not the name of a server.

Logging In

Most file servers require a user name and password for access. Medley maintains an ephemeral database of user names and passwords for each host accessed recently. The database vanishes when LOGOUT, SAVEVM, SYSOUT, or MAKESYS is executed, so that the passwords remain secure from any subsequent user of the same virtual memory image. Medley also maintains a notion of the "default" user name and password, which are generally those with which you initially log in.

When a file server for which the system does not yet have an entry in its password database requests a name and password, the system first tries the default user name and password. If the file server does not recognize that name/password, the system prompts you for a name and password to use for that host. It suggests a default name:

```
{ERIS} Login: Green
```

which you can accept by pressing [Return}, or replace the name by typing a new name or backspacing over it. Following the name, you are prompted for a password:

```
{ERIS} Login: Verdi (password)
```

which is not echoed, terminated by another [Return]. This information is stored in the password database so that you are prompted only once, until the database is again cleared.

Medley also prompts for password information when a protection violation occurs on accessing a directory on certain kinds of servers that support password-protected directories. Some such servers allow one to protect a file in a way that is inaccessible to even its owner until the file's protection is changed. In such cases, no password would help, and the system causes the normal PROTECTION VIOLATION error.

You can abort a password interaction by typing the ERROR interrupt, initially Cosntrol-E. This generally either causes a PROTECTION VIOLATION error, if the password was requested in order to gain access to a protected file on an otherwise accessible server; or to act as though the server did not exist, in the case where the password was needed to gain any access to the server.

(LOGIN *HOSTNAME FLG DIRECTORY MSG*) [Function]

Forces Medley to ask for your name and password to be used when accessing host *HOSTNAME*. Any previous login information for *HOSTNAME* is overridden. If *HOSTNAME* is NIL, it overrides login information for all hosts and resets the default user name and password to be those typed in by you. The special value *HOSTNAME* = NS:: is used to obtain the default user name and password for all logins for NS Servers.

If *FLG* is the atom QUIET, only prompts you if there is no cached information for *HOSTNAME*.

If *DIRECTORY* is specified, it is the name of a directory on *HOSTNAME*. In this case, the information requested is the "connect" password for that directory. Connect passwords for any number of different directories on a host can be maintained.

If *MSG* is non-NIL, it is a message (a string) to be printed before the name and password information is requested.

LOGIN returns the user name with which you completed the login.

(SETPASSWORD *HOST USER PASSWORD DIRECTORY*) [Function]

Sets the values in the internal password database exactly as if the strings *USER* and *PASSWORD* were typed in via (LOGIN *HOST* NIL *DIRECTORY*).

(SETUSERNAME *NAME*) [Function]

Sets the default uer name to *NAME*.

(USERNAME *FLG STRPTR PRESERVECASE*) [Function]

If *FLG* = NIL, returns the default user name. This is the only value of *FLG* that is meaningful in Medley.

USERNAME returns the value as a string, unless *STRPTR* is T, in which case USERNAME returns the value as an atom. The name is returned in uppercase, unless *PRESERVECASE* is true.

Abnormal Conditions

If Medley tries to access a file and does not get a response from the file server in a areasonable period of time, it prints a message that the file server is not responding, and keeps trying. If the file server has actually crashed, this may continue indefinitely. A Control-E or similar interrupt aborts out of this state.

If the file server crashes but is restarted before you attempt to do anything, file operations will usually proceed normally, except for a brief pause while Medley tries to re-establish any connections it had open before the crash. However, this is not always possible. For example, when a file is open for sequential output and the server crashes, there is no way to recover the output already written, since it vanished with the crash. In such cases, the system will cause an error such as `Connection Lost`.

`LOGOUT` closes any file server connections that are currently open. On return, it attempts to re-establish connections for any files that were open before logging out. If a file has disappeared or been modified, Medley reports this fact. Files that were open for sequential access generally cannot be reopened after `LOGOUT`.

Interlisp supports simultaneous access to the same server from different processes and permits overlapping of Lisp computation with file server operations, allowing for improved performance. However, as a corollary of this, a file is not closed the instant that `CLOSEF` returns; Interlisp closes the file "in the bckground". It is therefore very important that you exit Interlisp via `(LOGOUT)` or `(LOGOUT T)`, rather than boot the machine.

On rare occasions, the Ethernet may appear completely unresponsive, due to Interlisp having gotten into a bad state. Type `(RESTART.ETHER)` to reinitialize Lisp's Ethernet driver(s), just as when the Lisp system is started up following a `LOGOUT`, `SYSOUT`, etc.

24. INPUT/OUTPUT FUNCTIONS

This chapter describes the standard I/O functions used for reading and printing characters and Interlisp expressions on files and other streams. First, the primitive input functions are presented, then the output functions, then functions for random-access operations (such as searching a file for a given stream, or changing the "next-character" pointer to a position in a file). Next, the `PRINTOUT` statement is documented (see below), which provides an easy way to write complex output operations. Finally, read tables, used to parse characters as Interlisp expressions, are documented.

Specifying Streams for Input/Output Functions

Most of the input/output functions in Interlisp-D have an argument named *`STREAM`* or *`FILE`*, specifying on which open stream the function's action should occur (the name *`FILE`* is used in older functions that predate the concept of stream; the two should, however, be treated synonymously). The value of this argument should be one of the following:

a stream
: An object of type `STREAM`, as returned by `OPENSTREAM` (Chapter 23) or other stream-producing functions, is always the most precise and efficient way to designate a stream argument.

`T`
: The litatom `T` designates the terminal input or output stream of the currently running process, controlling input from the keyboard and output to the display screen. For functions where the direction (input or output) is ambiguous, `T` is taken to designate the terminal output stream. The `T` streams are always open; they cannot be closed.

  The terminal output stream can be set to a given window or display stream by using `TTYDISPLAYSTREAM` (Chapter 28). The terminal input stream cannot be changed. For more information on terminal I/O, see Chapter 30.

`NIL`
: The litatom `NIL` designates the "primary" input or output stream. These streams are initially the same as the terminal input/output streams, but they can be changed by using the functions `INPUT` and `OUTPUT`.

  For functions where the direction (input or output) is ambiguous, e.g., `GETFILEPTR`, the argument `NIL` is taken to mean the primary input stream, if that stream is not identical to the terminal input stream, else the primary output stream.

a window
: Uses the display stream of the window . Valid for output only.

a file name
: As of this writing, the name of an open file (as a litatom) can be used as a stream argument. However, there are inefficiencies and possible future incompatibilities associated with doing so. See Chapter 24 for details.

(**GETSTREAM** *FILE ACCESS*) [Function]

Coerces the argument *FILE* to a stream by the above rules. If *ACCESS* is INPUT, OUTPUT, or BOTH, produces the stream designated by *FILE* that is open for *ACCESS*. If *ACCESS*=NIL, returns a stream for *FILE* open for any kind of input/output (see the list above for the ambiguous cases). If *FILE* does not designate a stream open in the specified mode, causes an error, FILE NOT OPEN.

(**STREAMP** *X*) [Function]

Returns *X* if *X* is a STREAM, otherwise NIL.

Input Functions

While the functions described below can take input from any stream, some special actions occur when the input is from the terminal (the T input stream, see above). When reading from the terminal, the input is buffered a line at a time, unless buffering has been inhibited by CONTROL (Chapter 30) or the input is being read by READC or PEEKC. Using specified editing characters, you can erase a character at a time, a word at a time, or the whole line. The keys that perform these editing functions are assignable via SETSYNTAX, with the initial settings chosen to be those most natural for the given operating system. In Interlisp-D, the initial settings are as follows: characters are deleted one at a time by Backspace; words are erased by control-W; the whole line is erased by Control-Q.

On the Interlisp-D display, deleting a character or a line causes the characters to be physically erased from the screen. In Interlisp-10, the deleting action can be modified for various types of display terminals by using DELETECONTROL (Chapter 30).

Unless otherwise indicated, when the end of file is encountered while reading from a file, all input functions generate an error, END OF FILE. Note that this does not close the input file. The ENDOFSTREAMOP stream attribute (Chapter 24) is useful for changing the behavior at end of file.

Most input functions have a *RDTBL* argument, which specifies the read table to be used for input. Unless otherwise specified, if *RDTBL* is NIL, the primary read table is used.

If the *FILE* or *STREAM* argument to an input function is NIL, the primary input stream is used.

(**INPUT** *FILE*) [Function]

Sets *FILE* as the primary input stream; returns the old primary input stream. *FILE* must be open for input.

(INPUT) returns the current primary input stream, which is not changed.

Note: If the primary input stream is set to a file, the file's full name, rather than the stream itself, is returned. See discussion in Chapter 24.

(**READ** *FILE RDTBL FLG*) [Function]

Reads one expression from *FILE*. Atoms are delimited by the break and separator characters as defined in *RDTBL*. To include a break or separator character in an atom, the character must be preceded by the character %, e.g., AB%(C is the atom AB(C, %% is the atom %, %*control-K* is the atom Control-K. For input from the terminal, an atom containing an interrupt character can be input by typing instead the corresponding alphabetic character preceded by Control-V, e.g., ^VD for Control-D.

Strings are delimited by double quotes. To input a string containing a double quote or a %, precede it by %, e.g., "AB%"C" is the string AB"C. Note that % can always be typed even if next character is not "special", e.g., %A%B%C is read as ABC.

If an atom is interpretable as a number, READ creates a number, e.g., 1E3 reads as a floating point number, 1D3 as a literal atom, 1.0 as a number, 1,0 as a literal atom, etc. An integer can be input in a non-decimal radix by using syntax such as 123Q, |b10101, |5r1234 (see Chapter 7). The function RADIX, sets the radix used to print integers.

When reading from the terminal, all input is line-buffered to enable the action of the backspacing control characters, unless inhibited by CONTROL (Chapter 30). Thus no characters are actually seen by the program until a carriage-return (actually the character with terminal syntax class EOL, see Chapter 30), is typed. However, for reading by READ, when a matching right parenthesis is encountered, the effect is the same as though a carriage-return were typed, i.e., the characters are transmitted. To indicate this, Interlisp also prints a carriage-return line-feed on the terminal. The line buffer is also transmitted to READ whenever an IMMEDIATE read macro character is typed (see below).

FLG=T suppresses the carriage-return normally typed by READ following a matching right parenthesis. (However, the characters are still given to READ; i.e., you do not have to type the carriage-return.)

(**RATOM** *FILE RDTBL*) [Function]

Reads in one atom from *FILE*. Separation of atoms is defined by *RDTBL*. % is also defined for RATOM, and the remarks concerning line-buffering and editing control characters also apply.

If the characters comprising the atom would normally be interpreted as a number by READ, that number is returned by RATOM. Note however that RATOM takes no special action for " whether or not it is a break character, i.e., RATOM never makes a string.

(**RSTRING** *FILE RDTBL*) [Function]

Reads characters from *FILE* up to, but not including, the next break or separator character, and returns them as a string. Backspace, Control-W, Control-Q, Control-V, and % have the same effect as with READ.

Note that the break or separator character that terminates a call to RATOM or RSTRING is *not* read by that call, but remains in the buffer to become the first character seen by the next reading function that is called. If that function is RSTRING, it will return the null string. This is a common source of program bugs.

(**RATOMS** *A FILE RDTBL*) [Function]

Calls RATOM repeatedly until the atom *A* is read. Returns a list of the atoms read, not including *A*.

(**RATEST** *FLG*) [Function]

If *FLG* = T, RATEST returns T if a separator was encountered immediately prior to the atom returned by the last RATOM or READ, NIL otherwise.

If *FLG* = NIL, RATEST returns T if last atom read by RATOM or READ was a break character, NIL otherwise.

If *FLG* = 1, RATEST returns T if last atom read (by READ or RATOM) contained a % used to quote the next character (as in %[or %A%B%C), NIL otherwise.

(**READC** *FILE RDTBL*) [Function]

Reads and returns the next character, including %, ", etc, i.e., is not affected by break or separator characters. The action of READC is subject to line-buffering, i.e., READC does not return a value until the line has been terminated even if a character has been typed. Thus, the editing control characters have their usual effect. *RDTBL* does not directly affect the value returned, but is used as usual in line-buffering, e.g., determining when input has been terminated. If (CONTROL T) has been executed (Chapter 30), defeating line-buffering, the *RDTBL* argument is irrelevant, and READC returns a value as soon as a character is typed (even if the character typed is one of the editing characters, which ordinarily would never be seen in the input buffer).

(**PEEKC** *FILE*) [Function]

Returns the next character, but does not actually read it and remove it from the buffer. If reading from the terminal, the character is echoed as soon as PEEKC reads it, even though it is then "put back" into the system buffer, where Backspace, Control-W, etc. could change it. Thus it is possible for the value returned by PEEKC to "disagree" in the first character with a subsequent READ.

(**LASTC** *FILE*) [Function]

Returns the last character read from *FILE*. LASTC can return an incorrect result when called immediatley following a PEEKC on a file that contains run-coded NS characters.

(**READCCODE** *FILE RDTBL*) [Function]

Returns the next character *code* from *STREAM*; thus, this operation is equivalent to, but more efficient than, (CHCON1 (READC *FILE RDTBL*)).

(**PEEKCCODE** *FILE*) [Function]

Returns, without consuming, the next character *code* from *STREAM*; thus, this operation is equivalent to, but more efficient than, (CHCON1 (PEEKC *FILE*)).

(**BIN** *STREAM*) [Function]

Returns the next byte from *STREAM*. This operation is useful for reading streams of binary, rather than character, data.

Note: BIN is similar to READCCODE, except that BIN always reads a single byte, whereas READCCODE reads a "character" that can consist of more than one byte, depending on the character and its encoding.

READ, RATOM, RATOMS, PEEKC, READC all wait for input if there is none. The only way to test whether or not there is input is to use READP:

(**READP** *FILE FLG*) [Function]

Returns T if there is anything in the input buffer of *FILE*, NIL otherwise. This operation is only interesting for streams whose source of data is dynamic, e.g., the terminal or a byte stream over a network; for other streams, such as to files, (READP *FILE*) is equivalent to (NOT (EOFP *FILE*)).

Note that because of line-buffering, READP may return T, indicating there is input in the buffer, but READ may still have to wait.

Frequently, the terminal's input buffer contains a single EOL character left over from a previous input. For most applications, this situation wants to be treated as though the buffer were empty, and so READP returns NIL in this case. However, if *FLG*=T, READP returns T if there is *any* character in the input buffer, including a single EOL. *FLG* is ignored for streams other than the terminal.

(**EOFP** *FILE*) [Function]

Returns true if *FILE* is at "end of file", i.e., the next call to an input function would cause an END OF FILE error; NIL otherwise. For randomly accessible files, this can also be thought of as the file pointer pointing beyond the last byte of the file. *FILE* must be open for (at least) input, or an error is generated, FILE NOT OPEN.

Note that EOFP can return NIL and yet the next call to READ might still cause an END OF FILE error, because the only characters remaining in the input were separators or otherwise constituted an incomplete expression. The function SKIPSEPRS is sometimes more useful as a way of detecting end of file when it is known that all the expressions in the file are well formed.

(**WAITFORINPUT** *FILE*) [Function]

Waits until input is available from *FILE* or from the terminal, i.e. from T. WAITFORINPUT is functionally equivalent to (until (OR (READP T) (READP FILE)) do NIL), except that it does not use up machine cycles while waiting. Returns the device for which input is now available, i.e. *FILE* or T.

FILE can also be an integer, in which case WAITFORINPUT waits until there is input available from the terminal, or until *FILE* milliseconds have elapsed. Value is T if input is now available, NIL in the case that WAITFORINPUT timed out.

(**SKREAD** *FILE REREADSTRING RDTBL*) [Function]

"Skip Read". SKREAD consumes characters from *FILE* as if one call to READ had been performed, without paying the storage and compute cost to really read in the structure. *REREADSTRING* is for the case where the caller has already performed some READC's and RATOM's before deciding to skip this expression. In this case, *REREADSTRING* should be the material already read (as a string), and SKREAD operates as though it had seen that material first, thus setting up its parenthesis count, double-quote count, etc.

The read table *RDTBL* is used for reading from *FILE*. If *RDTBL* is NIL, it defaults to the value of FILERDTBL. SKREAD may have difficulties if unusual read macros are defined in *RDTBL*. SKREAD does not recognize read macro characters in *REREADSTRING*, nor SPLICE or INFIX read macros. This is only a problem if the read macros are defined to parse subsequent input in the stream that does not follow the normal parenthesis and string-quote conventions.

SKREAD returns %) if the read terminated on an unbalanced closing parenthesis; %] if the read terminated on an unbalanced %], i.e., one which also would have closed any extant open left parentheses; otherwise NIL.

(**SKIPSEPRS** *FILE RDTBL*) [Function]

Consumes characters from *FILE* until it encounters a non-separator character (as defined by *RDTBL*). SKIPSEPRS returns, but does not consume, the terminating character, so that the next call to READC would return the same character. If no non-separator character is found before the end of file is reached, SKIPSEPRS returns NIL and leaves the stream at end of file. This function is useful for skipping over "white space" when scanning a stream character by character, or for detecting end of file when reading expressions from a stream with no pre-arranged terminating expression.

Output Functions

Unless otherwise specified by DEFPRINT, pointers other than lists, strings, atoms, or numbers, are printed in the form {*DATATYPE*} followed by the octal representation of the address of the pointer (regardless of radix). For example, an array pointer might print as {ARRAYP}#43,2760. This printed representation is for compactness of display on your terminal, and will *not* read back in correctly; if the form above is read, it will produce the litatom {ARRAYP}#43,2760.

Note: The term "end-of-line" appearing in the description of an output function means the character or characters used to terminate a line in the file system being used by the given implementation of Interlisp. For example, in Interlisp-D end-of-line is indicated by the character carriage-return.

Some of the functions described below have a *RDTBL* argument, which specifies the read table to be used for output. If *RDTBL* is NIL, the primary read table is used.

Most of the functions described below have an argument *FILE*, which specifies the stream on which the operation is to take place. If *FILE* is NIL, the primary output stream is used .

(**OUTPUT** *FILE*) [Function]

Sets *FILE* as the primary output stream; returns the old primary output stream. *FILE* must be open for output.

(OUTPUT) returns the current primary output stream, which is not changed.

Note: If the primary output stream is set to a file, the file's full name, rather than the stream itself, is returned. See the discussion in Chapter 24.

(**PRIN1** *X FILE*) [Function]

Prints *X* on *FILE*.

(**PRIN2** *X FILE RDTBL*) [Function]

Prints *X* on *FILE* with %'s and "'s inserted where required for it to read back in properly by READ, using *RDTBL*.

Both PRIN1 and PRIN2 print any kind of Lisp expression, including lists, atoms, numbers, and strings. PRIN1 is generally used for printing expressions where human readability, rather than machine readability, is important, e.g., when printing text rather than program fragments. PRIN1 does not print double quotes around strings, or % in front of special characters. PRIN2 is used for printing Interlisp expressions which can then be read back into Interlisp with READ; i.e., break and separator characters in atoms will be preceded by %'s. For example, the atom "()" is printed as %(%) by PRIN2. If the integer output radix (as set by RADIX) is not 10, PRIN2 prints the integer using the input syntax for non-decimal integers (see Chapter 7) but PRIN1 does not (but both print the integer in the output radix).

(**PRIN3** *X FILE*) [Function]
(**PRIN4** *X FILE RDTBL*) [Function]

PRIN3 and PRIN4 are the same as PRIN1 and PRIN2 respectively, except that they do not increment the horizontal position counter nor perform any linelength checks. They are useful primarily for printing control characters.

(**PRINT** *X FILE RDTBL*) [Function]

Prints the expression *X* using PRIN2 followed by an end-of-line. Returns *X*.

(**PRINTCCODE** *CHARCODE FILE*) [Function]

Outputs a single character whose code is *CHARCODE* to *FILE*. This is similar to (PRIN1 (CHARACTER *CHARCODE*)), except that numeric characters are guaranteed to print "correctly"; e.g., (PRINTCCODE (CHARCODE 9)) always prints "9", independent of the setting of RADIX.

PRINTCCODE may actually print more than one byte on *FILE*, due to character encoding and end of line conventions; thus, no assumptions should be made about the relative motion of the file pointer (see GETFILEPTR) during this operation.

(**BOUT** *STREAM BYTE*) [Function]

Outputs a single 8-bit byte to *STREAM*. This is similar to PRINTCCODE, but for binary streams the character position in *STREAM* is not updated (as with PRIN3), and end of line conventions are ignored.

Note: BOUT is similar to PRINTCCODE, except that BOUT always writes a single byte, whereas PRINTCCODE writes a "character" that can consist of more than one byte, depending on the character and its encoding.

(**SPACES** *N FILE*) [Function]

Prints *N* spaces. Returns NIL.

(**TERPRI** *FILE*) [Function]

Prints an end-of-line character. Returns NIL.

(**FRESHLINE** *STREAM*) [Function]

Equivalent to TERPRI, except it does nothing if it is already at the beginning of the line. Returns T if it prints an end-of-line, NIL otherwise.

(**TAB** *POS MINSPACES FILE*) [Function]

Prints the appropriate number of spaces to move to position *POS*. *MINSPACES* indicates how many spaces must be printed (if NIL, 1 is used). If the current position plus *MINSPACES* is greater than *POS*, TAB does a TERPRI and then (SPACES *POS*). If *MINSPACES* is T, and the current position is greater than *POS*, then TAB does nothing.

Note: A sequence of PRINT, PRIN2, SPACES, and TERPRI expressions can often be more conveniently coded with a single PRINTOUT statement.

(**SHOWPRIN2** *X FILE RDTBL*) [Function]

Like PRIN2 except if SYSPRETTYFLG=T, prettyprints *X* instead. Returns *X*.

(**SHOWPRINT** *X FILE RDTBL*) [Function]

Like PRINT except if SYSPRETTYFLG=T, prettyprints *X* instead, followed by an end-of-line. Returns *X*.

SHOWPRINT and SHOWPRIN2 are used by the programmer's assistant (Chapter 13) for printing the values of expressions and for printing the history list, by various commands of the break package (Chapter 14), e.g. ?= and BT commands, and various other system packages. The idea is that by simply settting or binding SYSPRETTYFLG to T (initially NIL), you instruct the system when interacting with you to PRETTYPRINT expressions (Chapter 26) instead of printing them.

(**PRINTBELLS**) [Function]

Used by DWIM (Chapter 19) to print a sequence of bells to alert you to stop typing. Can be advised or redefined for special applications, e.g., to flash the screen on a display terminal.

(**FORCEOUTPUT** *STREAM WAITFORFINISH*) [Function]

Forces any buffered output data in *STREAM* to be transmitted.

If *WAITFORFINISH* is non-NIL, this doesn't return until the data has been forced out.

(**POSITION** *FILE N*) [Function]

Returns the column number at which the next character will be read or printed. After a end of line, the column number is 0. If *N* is non-NIL, *resets* the column number to be *N*.

Note that resetting POSITION only changes Lisp's belief about the current column number; it does not cause any horizontal motion. Also note that (POSITION *FILE*) is *not* the same as (GETFILEPTR *FILE*) which gives the position in the *file*, not on the *line*.

(**LINELENGTH** *N FILE*) [Function]

Sets the length of the print line for the output file *FILE* to *N*; returns the former setting of the line length. *FILE* defaults to the primary output stream. (LINELENGTH NIL *FILE*) returns the current setting for *FILE*. When a file is first opened, its line length is set to the value of the variable FILELINELENGTH.

Whenever printing an atom or string would increase a file's position *beyond* the line length of the file, an end of line is automatically inserted first. This action can be defeated by using PRIN3 and PRIN4.

(**SETLINELENGTH** *N*) [Function]

Sets the line length for the terminal by doing (LINELENGTH *N* T). If *N* is NIL, it determines *N* by consulting the operating system's belief about the terminal's characteristics. In Interlisp-D, this is a no-op.

PRINTLEVEL

When using Interlisp one often has to handle large, complicated lists, which are difficult to understand when printed out. PRINTLEVEL allows you to specify in how much detail lists should be printed. The print functions PRINT, PRIN1, and PRIN2 are all affected by level parameters set by:

(**PRINTLEVEL** *CARVAL CDRVAL*) [Function]

Sets the CAR print level to *CARVAL*, and the CDR print level to *CDRVAL*. Returns a list cell whose CAR and CDR are the old settings. PRINTLEVEL is initialized with the value (1000 . -1).

In order that PRINTLEVEL can be used with RESETFORM or RESETSAVE, if *CARVAL* is a list cell it is equivalent to (PRINTLEVEL (CAR *CARVAL*) (CDR *CARVAL*)).

(PRINTLEVEL *N* NIL) changes the CAR printlevel without affecting the CDR printlevel. (PRINTLEVEL NIL *N*) changes the CDR printlevel with affecting the CAR printlevel. (PRINTLEVEL) gives the current setting without changing either.

Note: Control-P (Chapter 30) can be used to change the PRINTLEVEL setting dynamically, even while Interlisp is printing.

The CAR printlevel specifies how "deep" to print a list. Specifically, it is the number of unpaired left parentheses which will be printed. Below that level, all lists will be printed as &. If the CAR printlevel is *negative*, the action is similar except that an end-of-line is inserted after each right parentheses that would be immediately followed by a left parenthesis.

The CDR printlevel specifies how "long" to print a list. It is the number of top level list elements that will be printed before the printing is terminated with --. For example, if *CDRVAL*=2, (A B C D E) will print as (A B --). For sublists, the number of list elements printed is also affected by the depth of printing in the CAR direction: Whenever the *sum* of the depth of the sublist (i.e. the number of unmatched left parentheses) and the number of elements is greater than the CDR printlevel, -- is printed. This gives a "triangular" effect in that less is printed the farther one goes in either CAR or CDR direction. If the CDR printlevel is negative, then it is the same as if the CDR printlevel were infinite.

Examples:

After:	(A (B C (D (E F) G) H) K L) prints as:
(PRINTLEVEL 3 -1)	(A (B C (D & G) H) K L)
(PRINTLEVEL 2 -1)	(A (B C & H) K L)
(PRINTLEVEL 1 -1)	(A & K L)
(PRINTLEVEL 0 -1)	&
(PRINTLEVEL 1000 2)	(A (B --) --)
(PRINTLEVEL 1000 3)	(A (B C --) K --)

```
(PRINTLEVEL 1 3)    (A & K --)
```

PLVLFILEFLG [Variable]

Normally, PRINTLEVEL only affects terminal output. Output to all other files acts as though the print level is infinite. However, if PLVLFILEFLG is T (initially NIL), then PRINTLEVEL affects output to files as well.

The following three functions are useful for printing isolated expressions at a specified print level without going to the overhead of resetting the global print level.

(**LVLPRINT** *X FILE CARLVL CDRLVL TAIL*) [Function]

Performs PRINT of *X* to *FILE*, using as CAR and CDR print levels the values *CARLVL* and *CDRLVL*, respectively. Uses the T read table. If *TAIL* is specified, and *X* is a tail of it, then begins its printing with "...", rather than on open parenthesis.

(**LVLPRIN2** *X FILE CARLVL CDRLVL TAIL*) [Function]

Similar to LVLPRIN2, but performs a PRIN2.

(**LVLPRIN1** *X FILE CARLVL CDRLVL TAIL*) [Function]

Similar to LVLPRIN1, but performs a PRIN1.

Printing Numbers

How the ordinary printing functions (PRIN1, PRIN2, etc.) print numbers can be affected in several ways. RADIX influences the printing of integers, and FLTFMT influences the printing of floating point numbers. The setting of the variable PRXFLG determines how the symbol-manipulation functions handle numbers. The PRINTNUM package permits greater controls on the printed appearance of numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

(**RADIX** *N*) [Function]

Resets the output radix for integers to the absolute value of *N*. The value of RADIX is its previous setting. (RADIX) gives the current setting without changing it. The initial setting is 10.

Note that RADIX affects output *only*. There is no input radix; on input, numbers are interpreted as decimal unless they are entered in a non-decimal radix with syntax such as 123Q, |b10101, |5r1234 (see Chapter 7). RADIX does not affect the behavior of UNPACK, etc., unless the value of PRXFLG (below) is T. For example, if PRXFLG is NIL and the radix is set to 8 with (RADIX 8), the value of (UNPACK 9) is (9), not (1 1).

Using PRINTNUM (below) or the PRINTOUT command .I (below) is often a more convenient and appropriate way to print a single number in a specified radix than to globally change RADIX.

(**FLTFMT** *FORMAT*) [Function]

Resets the output format for floating point numbers to the FLOAT format *FORMAT* (see PRINTNUM below for a description of FLOAT formats). *FORMAT*=T specifies the default "free" formatting: some number of significant digits (a function of the implementation) are printed, with trailing zeros suppressed; numbers with sufficiently large or small exponents are instead printed in exponent notation.

FLTFMT returns its current setting. (FLTFMT) returns the current setting without changing it. The initial setting is T.

Note: In Interlisp-D, FLTFMT ignores the *WIDTH* and *PAD* fields of the format (they are implemented only by PRINTNUM).

Whether print name manipulation functions (UNPACK, NCHARS, etc.) use the values of RADIX and FLTFMT is determined by the variable PRXFLG:

PRXFLG [Variable]

If PRXFLG=NIL (the initial setting), then the "PRIN1" name used by PACK, UNPACK, MKSTRING, etc., is computed using base 10 for integers and the system default floating format for floating point numbers, independent of the current setting of RADIX or FLTFMT. If PRXFLG=T, then RADIX and FLTFMT do dictate the "PRIN1" name of numbers. Note that in this case, PACK and UNPACK are *not* inverses.

Examples with (RADIX 8), (FLTFMT '(FLOAT 4 2)):

With PRXFLG=NIL,

```
(UNPACK 13)  =>  (1 3)
(PACK '(A 9))  =>  A9
(UNPACK 1.2345)  =>  (1 %. 2 3 4 5)
```

With PRXFLG=T,

```
(UNPACK 13)  =>  (1 5)
(PACK '(A 9))  =>  A11
(UNPACK 1.2345)  =>  (1 %. 2 3)
```

Note that PRXFLG does not effect the radix of "PRIN2" names, so with (RADIX 8), (NCHARS 9 T), which uses PRIN2 names, would return 3, (since 9 would print as 11Q) for either setting of PRXFLG.

Warning: Some system functions will not work correctly if PRXFLG is not NIL. Therefore, resetting the global value of PRXFLG is not recommended. It is much better to rebind PRXFLG as a SPECVAR for that part of a program where it needs to be non-NIL.

The basic function for printing numbers under format control is PRINTNUM. Its utility is considerably enhanced when used in conjunction with the PRINTOUT package, which implements a compact language for specifying complicated sequences of elementary printing operations, and makes fancy output formats easy to design and simple to program.

(**PRINTNUM** FORMAT NUMBER FILE) [Function]

Prints *NUMBER* on *FILE* according to the format *FORMAT*. *FORMAT* is a list structure with one of the forms described below.

If *FORMAT* is a list of the form (FIX *WIDTH RADIX PAD0 LEFTFLUSH*), this specifies a FIX format. *NUMBER* is rounded to the nearest integer, and then printed in a field *WIDTH* characters long with radix set to *RADIX* (or 10 if *RADIX*=NIL; note that the setting from the function RADIX is *not* used as the default). If *PAD0* and *LEFTFLUSH* are both NIL, the number is right-justified in the field, and the padding characters to the left of the leading digit are spaces. If *PAD0* is T, the character "0" is used for padding. If *LEFTFLUSH* is T, then the number is left-justified in the field, with trailing spaces to fill out *WIDTH* characters.

The following examples illustrate the effects of the FIX format options on the number 9 (the vertical bars indicate the field width):

FORMAT:	(PRINTNUM *FORMAT* 9) prints:
(FIX 2)	` 9`
(FIX 2 NIL T)	`09`
(FIX 12 8 T)	`000000000011`
(FIX 5 NIL NIL T)	`9    `

If *FORMAT* is a list of the form (FLOAT *WIDTH DECPART EXPPART PAD0 ROUND*), this specifies a FLOAT format. *NUMBER* is printed as a decimal number in a field *WIDTH* characters wide, with *DECPART* digits to the right of the decimal point. If *EXPPART* is not 0 (or NIL), the number is printed in exponent notation, with the exponent occupying *EXPPART* characters in the field. *EXPPART* should allow for the character E and an optional sign to be printed before the exponent digits. As with FIX format, padding on the left is with spaces, unless *PAD0* is T. If *ROUND* is given, it indicates the digit position at which rounding is to take place, counting from the leading digit of the number.

Interlisp-D interprets *WIDTH*=NIL to mean no padding, i.e., to use however much space the number needs, and interprets *DECPART*=NIL to mean as many decimal places as needed.

The following examples illustrate the effects of the FLOAT format options on the number 27.689 (the vertical bars indicate the field width):

FORMAT:	(PRINTNUM *FORMAT* 27.689) prints:
(FLOAT 7 2)	`  27.69`
(FLOAT 7 2 NIL 0)	`0027.69`
(FLOAT 7 2 2)	` 2.77E1`
(FLOAT 11 2 4)	`   2.77E+01`
(FLOAT 7 2 NIL NIL 1)	`  30.00`
(FLOAT 7 2 NIL NIL 2)	`  28.00`

NILNUMPRINTFLG [Variable]

If PRINTNUM's *NUMBER* argument is not a number and not NIL, a NON-NUMERIC ARG error is generated. If *NUMBER* is NIL, the effect depends on the setting of the variable NILNUMPRINTFLG. If NILNUMPRINTFLG is NIL, then the error occurs as usual. If it is non-NIL, then no error occurs, and the value of NILNUMPRINTFLG is printed right-justified in the field described by *FORMAT*. This option facilitates the printing of numbers in aggregates with missing values coded as NIL.

User Defined Printing

Initially, Interlisp only knows how to print in an interesting way objects of type litatom, number, string, list and stackp. All other types of objects are printed in the form {datatype} followed by the octal representation of the address of the pointer, a format that cannot be read back in to produce an equivalent object. When defining user data types (using the DATATYPE record type, Chapter 8), it is often desirable to specify as well how objects of that type should be printed, so as to make their contents readable, or at least more informative to the viewer. The function DEFPRINT is used to specify the printing format of a data type.

(**DEFPRINT** *TYPE FN*) [Function]

TYPE is a type name. Whenever a printing function (PRINT, PRIN1, PRIN2, etc.) or a function requiring a print name (CHCON, NCHARS, etc.) encounters an object of the indicated type, *FN* is called with two arguments: the item to be printed and the name of the stream, if any, to which the object is to be printed. The second argument is NIL on calls that request the print name of an object without actually printing it.

If *FN* returns a list of the form (*ITEM1* . *ITEM2*), *ITEM1* is printed using PRIN1 (unless it is NIL), and then *ITEM2* is printed using PRIN2 (unless it is NIL). No spaces are printed between the two items. Typically, *ITEM1* is a read macro character.

If *FN* returns NIL, the datum is printed in the system default manner.

If *FN* returns T, nothing further is printed; *FN* is assumed to have printed the object to the stream itself. Note that this case if permitted only when the second argument passed to *FN* is non-NIL; otherwise, there is no destination for *FN* to do its printing, so it must return as in one of the other two cases.

Printing Unusual Data Structures

HPRINT (for "Horrible Print") and HREAD provide a mechanism for printing and reading back in general data structures that cannot normally be dumped and loaded easily, such as (possibly re-entrant or circular) structures containing user datatypes, arrays, hash tables, as well as list structures. HPRINT will correctly print and read back in any structure containing any or all of the above, chasing all pointers down to the level of literal atoms, numbers or strings. HPRINT currently cannot handle compiled code arrays, stack positions, or arbitrary unboxed numbers.

HPRINT operates by simulating the Interlisp PRINT routine for normal list structures. When it encounters a user datatype (see Chapter 8), or an array or hash array, it prints the data contained therein, surrounded by special characters defined as read macro characters. While chasing the pointers of a structure, it also keeps a hash table of those items it encounters, and if any item is encountered a second time, another read macro character is inserted before the first occurrence (by resetting the file pointer with SETFILEPTR) and all subsequent occurrences are printed as a back reference using an appropriate macro character. Thus the inverse function, HREAD merely calls the Interlisp READ routine with the appropriate read table.

(**HPRINT** *EXPR FILE UNCIRCULAR DATATYPESEEN*) [Function]

Prints *EXPR* on *FILE*. If *UNCIRCULAR* is non-NIL, HPRINT does no checking for any circularities in *EXPR* (but is still useful for dumping arbitrary structures of arrays, hash arrays, lists, user data types, etc., that do not contain circularities). Specifying *UNCIRCULAR* as non-NIL results in a large speed and internal-storage advantage.

Normally, when HPRINT encounters a user data type for the first time, it outputs a summary of the data type's declaration. When this is read in, the data type is redeclared. If *DATATYPESEEN* is non-NIL, HPRINT assumes that the same data type declarations will be in force at read time as were at HPRINT time, and not output declarations.

HPRINT is intended primarily for output to random access files, since the algorithm depends on being able to reset the file pointer. If *FILE* is not a random access file (and *UNCIRCULAR* = NIL), a temporary file, HPRINT.SCRATCH, is opened, *EXPR* is HPRINTed on it, and then that file is copied to the final output file and the temporary file is deleted.

You can not use HPRINT to save things that contains pointers to raw storage. Fontdescriptors contain pointers to raw storage and windows contain pointers to fontdescriptors. Netiher can therefor be saved with HPRINT.

(**HREAD** *FILE*) [Function]

Reads and returns an HPRINT-ed expression from *FILE*.

(**HCOPYALL** *X*) [Function]

Copies data structure *X*. *X* may contain circular pointers as well as arbitrary structures.

Note: HORRIBLEVARS and UGLYVARS (Chapter 17) are two file package commands for dumping and reloading circular and re-entrant data structures. They provide a convenient interface to HPRINT and HREAD.

When HPRINT is dumping a data structure that contains an instance of an Interlisp datatype, the datatype declaration is also printed onto the file. Reading such a data structure with HREAD can cause problems if it redefines a system datatype. Redefining a system datatype will almost definitely cause serious errors. The Interlisp system datatypes do not change very often, but there is always a possibility when loading in old files created under an old Interlisp release.

To prevent accidental system crashes, HREAD will *not* redefine datatypes. Instead, it will cause an error "attempt to read DATATYPE with different field `specification than currently defined"`. Continuing from this error will redefine the datatype.

Random Access File Operations

For most applications, files are read starting at their beginning and proceeding sequentially, i.e., the next character read is the one immediately following the last character read. Similarly, files are written sequentially. However, for files on some devices, it is also possible to read / write characters at arbitrary positions in a file, essentially treating the file as a large block of auxiliary storage. For example, one application might involve writing an expression at the *beginning* of the file, and then reading an expression from a specified point in its *middle*. This particular example requires the file be open for *both* input and output. However, random file input or output can also be performed on files that have been opened for only input or only output.

Associated with each file is a "file pointer" that points to the location where the next character is to be read from or written to. The file position of a byte is the number of bytes that precede it in the file, i.e., 0 is the position of the beginning of the file. The file pointer to a file is automatically advanced after each input or output operation. This section describes functions which can be used to *reposition* the file pointer on those files that can be randomly accessed. A file used in this fashion is much like an array in that it has a certain number of addressable locations that characters can be put into or taken from. However, unlike arrays, files can be enlarged. For example, if the file pointer is positioned at the end of a file and anything is written, the file "grows." It is also possible to position the file pointer *beyond* the end of file and then to write. (If the program attempts to *read* beyond the end of file, an `END OF FILE` error occurs.) In this case, the file is enlarged, and a "hole" is created, which can later be written into. Note that this enlargement only takes place at the *end* of a file; it is not possible to make more room in the middle of a file. In other words, if expression `A` begins at position 1000, and expression `B` at 1100, and the program attempts to overwrite `A` with expression `C`, whose printed representation is 200 bytes long, part of `B` will be altered.

Warning: File positions are always in terms of bytes, not characters. You should thus be very careful about computing the space needed for an expression. In particular, NS characters may take multiple bytes (see below). Also, the end-of-line character (see Chapter 24) may be represented by a different number of characters in different implementations. Output functions may also introduce end-of-line’s as a result of `LINELENGTH` considerations. Therefore `NCHARS` (see Chapter 2) does *not* specify how many bytes an expression takes to print, even ignoring line length considerations.

(**GETFILEPTR** *FILE*) [Function]

Returns the current position of the file pointer for *FILE*, i.e., the byte address at which the next input / output operation will commence.

(**SETFILEPTR** *FILE ADR*) [Function]

Sets the file pointer for *FILE* to the position *ADR*; returns *ADR*. The special value *ADR*`=-1` is interpreted to mean the address of the end of file.

Note: If a file is opened for output only, the end of file is initially zero, even if an old file by the same name had existed (see OPENSTREAM, Chapter 24). If a file is opened for both input and output, the initial file pointer is the beginning of the file, but (SETFILEPTR *FILE* -1) sets it to the end of the file. If the file had been opened in append mode by (OPENSTREAM *FILE* 'APPEND), the file pointer right after opening would be set to the end of the existing file, in which case a SETFILEPTR to position the file at the end would be unnecessary.

(**GETEOFPTR** *FILE*) [Function]

Returns the byte address of the end of file, i.e., the number of bytes in the file. Equivalent to performing (SETFILEPTR *FILE* -1) and returning (GETFILEPTR *FILE*) except that it does not change the current file pointer.

(**RANDACCESSP** *FILE*) [Function]

Returns *FILE* if *FILE* is randomly accessible, NIL otherwise. The file T is not randomly accessible, nor are certain network file connections in Interlisp-D. *FILE* must be open or an error is generated, FILE NOT OPEN.

(**COPYBYTES** *SRCFIL DSTFIL START END*) [Function]

Copies bytes from *SRCFIL* to *DSTFIL*, starting from position *START* and up to but not including position *END*. Both *SRCFIL* and *DSTFIL* must be open. Returns T.

If *END*=NIL, *START* is interpreted as the number of bytes to copy (starting at the current position). If *START* is also NIL, bytes are copied until the end of the file is reached.

Warning: COPYBYTES does not take any account of multi-byte NS characters (see Chapter 2). COPYCHARS (below) should be used whenever copying information that might include NS characters.

(**COPYCHARS** *SRCFIL DSTFIL START END*) [Function]

Like COPYBYTES except that it copies NS characters (see Chapter 2), and performs the proper conversion if the end-of-line conventions of *SRCFIL* and *DSTFIL* are not the same (see Chapter 24). *START* and *END* are interpreted the same as with COPYBYTES, i.e., as byte (not character) specifications in *SRCFIL*. The number of bytes actually output to *DSTFIL* might be more or less than the number of bytes specified by *START* and *END*, depending on what the end-of-line conventions are. In the case where the end-of-line conventions happen to be the same, COPYCHARS simply calls COPYBYTES.

(**FILEPOS** *STR FILE START END SKIP TAIL CASEARRAY*) [Function]

Analogous to STRPOS (see Chapter 4), but searches a file rather than a string. FILEPOS searches *FILE* for the string *STR*. Search begins at *START* (or the current position of the file pointer, if *START*=NIL), and goes to *END* (or the end of *FILE*, if *END*=NIL). Returns the address of the start of the match, or NIL if not found.

SKIP can be used to specify a character which matches any character in the file. If *TAIL* is T, and the search is successful, the value is the address of the first character *after* the sequence of characters corresponding to *STR*, instead of the starting address of the sequence. In either case, the file is left so that the next i/o operation begins at the address returned as the value of FILEPOS.

CASEARRAY should be a "case array" that specifies that certain characters should be transformed to other characters before matching. Case arrays are returned by CASEARRAY or SEPRCASE below. *CASEARRAY*=NIL means no transformation will be performed.

A case array is an implementation-dependent object that is logically an array of character codes with one entry for each possible character. FILEPOS maps each character in the file "through" *CASEARRAY* in the sense that each character code is transformed into the corresponding character code from *CASEARRAY* before matching. Thus if two characters map into the same value, they are treated as equivalent by FILEPOS. CASEARRAY and SETCASEARRAY provide an implementation-independent interface to case arrays.

For example, to search without regard to upper and lower case differences, *CASEARRAY* would be a case array where all characters map to themselves, except for lower case characters, whose corresponding elements would be the upper case characters. To search for a delimited atom, one could use " *ATOM* " as the pattern, and specify a case array in which all of the break and separator characters mapped into the same code as space.

For applications calling for extensive file searches, the function FFILEPOS is often faster than FILEPOS.

(**FFILEPOS** *PATTERN FILE START END SKIP TAIL CASEARRAY*) [Function]

Like FILEPOS, except much faster in most applications. FFILEPOS is an implementation of the Boyer-Moore fast string searching algorithm. This algorithm preprocesses the string being searched for and then scans through the file in steps usually equal to the length of the string. Thus, FFILEPOS speeds up roughly in proportion to the length of the string, e.g., a string of length 10 will be found twice as fast as a string of length 5 in the same position.

Because of certain fixed overheads, it is generally better to use FILEPOS for short searches or short strings.

(**CASEARRAY** *OLDARRAY*) [Function]

Creates and returns a new case array, with all elements set to themselves, to indicate the identity mapping. If *OLDARRAY* is given, it is reused.

(**SETCASEARRAY** *CASEARRAY FROMCODE TOCODE*) [Function]

Modifies the case array *CASEARRAY* so that character code *FROMCODE* is mapped to character code *TOCODE*.

(**GETCASEARRAY** *CASEARRAY FROMCODE*) [Function]

Returns the character code that *FROMCODE* is mapped to in *CASEARRAY*.

(**SEPRCASE** *CLFLG*) [Function]

Returns a new case array suitable for use by FILEPOS or FFILEPOS in which all of the break/separators of FILERDTBL are mapped into character code zero. If *CLFLG* is non-NIL, then all CLISP characters are mapped into this character as well. This is useful for finding a delimited atom in a file. For example, if *PATTERN* is " FOO ", and (SEPRCASE T) is used for *CASEARRAY*, then FILEPOS will find "(FOO_".

UPPERCASEARRAY [Variable]

Value is a case array in which every lowercase character is mapped into the corresponding uppercase character. Useful for searching text files.

Input/Output Operations with Characters and Bytes

Interlisp-D supports the 16-bit NS character set (see Chapter 2). All of the standard string and print name functions accept litatoms and strings containing NS characters. In almost all cases, a program does not have to distinguish between NS characters or 8-bit characters. The exception to this rule is the handling of input/output operations.

Interlisp-D uses two ways of writing 16-bit NS characters on files. One way is to write the full 16-bits (two bytes) every time a character is output. The other way is to use "run-encoding." Each 16 NS character can be decoded into a character set (an integer from 0 to 254 inclusive) and a character number (also an integer from 0 to 254 inclusive). In run-encoding, the byte 255 (illegal as either a character set number or a character number) is used to signal a change to a given character set, and the following bytes are all assumed to come from the same character set (until the next change-character set sequence). Run-encoding can reduce the number of bytes required to encode a string of NS characters, as long as there are long sequences of characters from the same character set (usually the case).

Note that characters are not the same as bytes. A single character can take anywhere from one to four bytes bytes, depending on whether it is in the same character set as the preceeding character, and whether run-encoding is enabled. Programs which assume that characters are equal to bytes must be changed to work with NS characters.

The functions BIN and BOUT (see above) should only be used to read and write single eight-bit bytes. The functions READCCODE and PRINTCCODE (see above) should be used to read and write single character codes, interpreting run-encoded NS characters. COPYBYTES should only be used to copy blocks of 8-bit data; COPYCHARS should be used to copy characters. Most I/O functions (READC, PRIN1, etc.) read or write 16-bit NS characters.

The use of NS characters has serious consequences for any program that uses file pointers to access a file in a random access manner. At any point when a file is being read or written, it has a "current character set." If the file pointer is changed with SETFILEPTR to a part of the file with a different character set, any characters read or written may have the wrong character set. The current character set can be accessed with the following function:

(**CHARSET** *STREAM CHARACTERSET*) [Function]

Returns the current character set of the stream *STREAM*. If *CHARACTERSET* is non-NIL, the current character set for *STREAM* is set. Note that for output streams this may cause bytes to be written to the stream.

If *CHARACTERSET* is T, run encoding for *STREAM* is disabled: both the character set and the character number (two bytes total) will be written to the stream for each character printed.

PRINTOUT

Interlisp provides many facilities for controlling the format of printed output. By executing various sequences of PRIN1, PRIN2, TAB, TERPRI, SPACES, PRINTNUM, and PRINTDEF, almost any effect can be achieved. PRINTOUT implements a compact language for specifying complicated sequences of these elementary printing functions. It makes fancy output formats easy to design and simple to program.

PRINTOUT is a CLISP word (like FOR and IF) for interpreting a special printing language in which you can describe the kinds of printing desired. The description is translated by DWIMIFY to the appropriate sequence of PRIN1, TAB, etc., before it is evaluated or compiled. PRINTOUT printing descriptions have the following general form:

(PRINTOUT *STREAM* $PRINTCOM_1$ . . . $PRINTCOM_N$)

STREAM is evaluated to obtain the stream to which the output from this specification is directed. The PRINTOUT commands are strung together, one after the other without punctuation, after *STREAM*. Some commands occupy a single position in this list, but many commands expect to find arguments following the command name in the list. The commands fall into several logical groups: one set deals with horizontal and vertical spacing, another group provides controls for certain formatting capabilities (font changes and subscripting), while a third set is concerned with various ways of actually printing items. Finally, there is a command that permits escaping to a simple Lisp evaluation in the middle of a PRINTOUT form. The various commands are described below. The following examples give a general flavor of how PRINTOUT is used:

Example 1: Suppose you want to print out on the terminal the values of three variables, X, Y, and Z, separated by spaces and followed by a carriage return. This could be done by:

```
(PRIN1 X T)
       (SPACES 1 T)
       (PRIN1 Y T)
       (SPACES 1 T)
       (PRIN1 Z T)
       (TERPRI T)
```

or by the more concise PRINTOUT form:

```
(PRINTOUT T X , Y , Z T)
```

Here the first T specifies output to the terminal, the commas cause single spaces to be printed, and the final T specifies a TERPRI. The variable names are not recognized as special PRINTOUT commands, so they are printed using PRIN1 by default.

Example 2: Suppose the values of X and Y are to be pretty-printed lined up at position 10, preceded by identifying strings. If the output is to go to the primary output stream, you could write either:

```
(PRIN1 "X =")
       (PRINTDEF X 10 T)
       (TERPRI )
       (PRIN1 "Y =")
       (PRINTDEF Y 10 T)
       (TERPRI)
```

or the equivalent:

```
(PRINTOUT NIL "X =" 10 .PPV X T
          "Y =" 10 .PPV Y T)
```

Since strings are not recognized as special commands, "X =" is also printed with PRIN1 by default. The positive integer means TAB to position 10, where the .PPV command causes the value of X to be prettyprinted as a variable. By convention, special atoms used as PRINTOUT commands are prefixed with a period. The T causes a carriage return, so the Y information is printed on the next line.

Example 3. As a final example, suppose that the value of X is an integer and the value of Y is a floating-point number. X is to be printed right-flushed in a field of width 5 beginning at position 15, and Y is to be printed in a field of width 10 also starting at position 15 with 2 places to the right of the decimal point. Furthermore, suppose that the variable names are to appear in the font class named BOLDFONT and the values in font class SMALLFONT. The program in ordinary Interlisp that would accomplish these effects is too complicated to include here. With PRINTOUT, one could write:

```
(PRINTOUT NIL
          .FONT BOLDFONT "X =" 15
```

```
.FONT SMALLFONT .I5 X T
.FONT BOLDFONT "Y =" 15
.FONT SMALLFONT .F10.2 Y T
.FONT BOLDFONT)
```

The .FONT commands do whatever is necessary to change the font on a multi-font output device. The .I5 command sets up a FIX format for a call to the function PRINTNUM (see above) to print X in the desired format. The .F10.2 specifies a FLOAT format for PRINTNUM.

Horizontal Spacing Commands

The horizontal spacing commands provide convenient ways of calling TAB and SPACES. In the following descriptions, *N* stands for a literal positive integer (*not* for a variable or expression whose value is an integer).

N (*N* a number) [PRINTOUT Command]

Used for absolute spacing. It results in a TAB to position *N* (literally, a (TAB *N*)). If the line is currently at position *N* or beyond, the file will be positioned at position *N* on the next line.

.TAB *POS* [PRINTOUT Command]

Specifies TAB to position (the value of) *POS*. This is one of several commands whose effect could be achieved by simply escaping to Lisp, and executing the corresponding form. It is provided as a separate command so that the PRINTOUT form is more concise and is prettyprinted more compactly. Note that .TAB *N* and *N*, where *N* is an integer, are equivalent.

.TAB0 *POS* [PRINTOUT Command]

Like .TAB except that it can result in zero spaces (i.e. the call to TAB specifies *MINSPACES*=0).

-N (*N* a number) [PRINTOUT Command]

Negative integers indicate relative (as opposed to absolute) spacing. Translates as (SPACES |*N*|).

, [PRINTOUT Command]
,, [PRINTOUT Command]
,,, [PRINTOUT Command]

(1, 2 or 3 commas) Provides a short-hand way of specifying 1, 2 or 3 spaces, i.e., these commands are equivalent to -1, -2, and -3, respectively.

.SP *DISTANCE* [PRINTOUT Command]

Translates as (SPACES *DISTANCE*). Note that .SP *N* and -*N*, where *N* is an integer, are equivalent.

Vertical Spacing Commands

Vertical spacing is obtained by calling TERPRI or printing form-feeds. The relevant commands are:

T [PRINTOUT Command]

Translates as (TERPRI), i.e., move to position 0 (the first column) of the next line. To print the letter T, use the string "T".

.SKIP *LINES* [PRINTOUT Command]

Equivalent to a sequence of *LINES* (TERPRI)'s. The .SKIP command allows for skipping large constant distances and for computing the distance to be skipped.

.PAGE [PRINTOUT Command]

Puts a form-feed (Control-L) out on the file. Care is taken to make sure that Interlisp's view of the current line position is correctly updated.

Special Formatting Controls

There are a small number of commands for invoking some of the formatting capabilities of multi-font output devices. The available commands are:

.FONT *FONTSPEC* [PRINTOUT Command]

Changes printing to the font *FONTSPEC*, which can be a font descriptor, a "font list" such as '(MODERN 10), an image stream (coerced to its current font), or a windows (coerced to the current font of its display stream). The DSPFONT is changed permanently. See fonts (Chapter 27) for more information.

FONTSPEC may also be a positive integer *N*, which is taken as an abbreviated reference to the font class named FONT*N* (e.g. 1 => FONT1).

.SUP [PRINTOUT Command]

Specifies superscripting. All subsequent characters are printed above the base of the current line. Note that this is absolute, not relative: a .SUP following a .SUP is a no-op.

.SUB [PRINTOUT Command]

Specifies subscripting. Subsequent printing is below the base of the current line. As with superscripting, the effect is absolute.

.BASE [PRINTOUT Command]

Moves printing back to the base of the current line. Un-does a previous .SUP or .SUB; a no-op, if printing is currently at the base.

Printing Specifications

The value of any expression in a PRINTOUT form that is not recognized as a command itself or as a command argument is printed using PRIN1 by default. For example, title strings can be printed by simply including the string as a separate PRINTOUT command, and the values of variables and forms can be printed in much the same way. Note that a literal integer, say 51, cannot be printed by including it as a command, since it would be interpreted as a TAB; the desired effect can be obtained by using instead the string specification "51", or the form (QUOTE 51).

For those instances when PRIN1 is not appropriate, e.g., PRIN2 is required, or a list structures must be prettyprinted, the following commands are available:

.P2 *THING* [PRINTOUT Command]

Causes *THING* to be printed using PRIN2; translates as (PRIN2 *THING*).

.PPF *THING* [PRINTOUT Command]

Causes *THING* to be prettyprinted at the current line position via PRINTDEF (see Chapter 26). The call to PRINTDEF specifies that *THING* is to be printed as if it were part of a function definition. That is, SELECTQ, PROG, etc., receive special treatment.

.PPV *THING* [PRINTOUT Command]

Prettyprints *THING* as a variable; no special interpretation is given to SELECTQ, PROG, etc.

.PPFTL *THING* [PRINTOUT Command]

Like .PPF, but prettyprints *THING* as a *tail*, that is, without the initial and final parentheses if it is a list. Useful for prettyprinting sub-lists of a list whose other elements are formatted with other commands.

.PPVTL *THING* [PRINTOUT Command]

Like .PPV, but prettyprints *THING* as a tail.

Paragraph Format

Interlisp's prettyprint routines are designed to display the structure of expressions, but they are not really suitable for formatting unstructured text. If a list is to be printed as a textual paragraph, its internal structure is less important than controlling its left and right margins, and the indentation of its first line. The .PARA and .PARA2 commands allow these parameters to be conveniently specified.

.PARA *LMARG RMARG LIST* [PRINTOUT Command]

Prints *LIST* in paragraph format, using PRIN1. Translates as (PRINTPARA *LMARG RMARG LIST*) (see below).

Example: `(PRINTOUT T 10 .PARA 5 -5 LST)` will print the elements of `LST` as a paragraph with left margin at 5, right margin at `(LINELENGTH)`-5, and the first line indented to 10.

.PARA2 *LMARG RMARG LIST* [PRINTOUT Command]

Print as paragraph using `PRIN2` instead of `PRIN1`. Translates as `(PRINTPARA` *LMARG RMARG LIST* `T)`.

Right-Flushing

Two commands are provided for printing simple expressions flushed-right against a specified line position, using the function `FLUSHRIGHT` (see below). They take into account the current position, the number of characters in the print-name of the expression, and the position the expression is to be flush against, and then print the appropriate number of spaces to achieve the desired effect. Note that this might entail going to a new line before printing. Note also that right-flushing of expressions longer than a line (e.g. a large list) makes little sense, and the appearance of the output is not guaranteed.

.FR *POS EXPR* [PRINTOUT Command]

Flush-right using `PRIN1`. The value of *POS* determines the position that the right end of *EXPR* will line up at. As with the horizontal spacing commands, a negative position number means |*POS*| columns from the current position, a positive number specifies the position absolutely. *POS*=0 specifies the right-margin, i.e. is interpreted as `(LINELENGTH)`.

.FR2 *POS EXPR* [PRINTOUT Command]

Flush-right using `PRIN2` instead of `PRIN1`.

Centering

Commands for centering simple expressions between the current line position and another specified position are also available. As with right flushing, centering of large expressions is not guaranteed.

.CENTER *POS EXPR* [PRINTOUT Command]

Centers *EXPR* between the current line position and the position specified by the value of *POS*. A positive *POS* is an absolute position number, a negative *POS* specifies a position relative to the current position, and 0 indicates the right-margin. Uses `PRIN1` for printing.

.CENTER2 *POS EXPR* [PRINTOUT Command]

Centers using `PRIN2` instead of `PRIN1`.

Numbering

The following commands provide FORTRAN-like formatting capabilities for integer and floating-point numbers. Each command specifies a printing format and a number to be printed. The format specification translates into a format-list for the function PRINTNUM.

.I*FORMAT NUMBER* [PRINTOUT Command]

Specifies integer printing. Translates as a call to the function PRINTNUM with a FIX format-list constructed from *FORMAT*. The atomic format is broken apart at internal periods to form the format-list. For example, .I5.8.T yields the format-list (FIX 5 8 T), and the command sequence (PRINTOUT T .I5.8.T FOO) translates as (PRINTNUM '(FIX 5 8 T) FOO). This expression causes the value of FOO to be printed in radix 8 right-flushed in a field of width 5, with 0's used for padding on the left. Internal NIL's in the format specification may be omitted, e.g., the commands .I5..T and .I5.NIL.T are equivalent.

The format specification .I1 is often useful for forcing a number to be printed in radix 10 (but not otherwise specially formatted), independent of the current setting of RADIX.

.F *FORMAT NUMBER* [PRINTOUT Command]

Specifies floating-number printing. Like the .I format command, except translates with a FLOAT format-list.

.N *FORMAT NUMBER* [PRINTOUT Command]

The .I and .F commands specify calls to PRINTNUM with quoted format specifications. The .N command translates as (PRINTNUM *FORMAT NUMBER*), i.e., it permits the format to be the value of some expression. Note that, unlike the .I and .F commands, *FORMAT* is a separate element in the command list, not part of an atom beginning with .N.

Escaping to Lisp

There are many reasons for taking control away from PRINTOUT in the middle of a long printing expression. Common situations involve temporary changes to system printing parameters (e.g. LINELENGTH), conditional printing (e.g. print FOO only if FIE is T), or lower-level iterative printing within a higher-level print specification.

*FORM* [PRINTOUT Command]

The escape command. *FORM* is an arbitrary Lisp expression that is evaluated within the context established by the PRINTOUT form, i.e., *FORM* can assume that the primary output stream has been set to be the *FILE* argument to PRINTOUT. Note that nothing is done with the *value* of *FORM*; any printing desired is accomplished by *FORM* itself, and the value is discarded.

Note: Although PRINTOUT logically encloses its translation in a RESETFORM (Chapter 14) to change the primary output file to the *FILE* argument (if non-NIL), in most cases it can actually pass *FILE* (or a locally bound variable if *FILE* is a non-trivial expression) to each printing function. Thus, the RESETFORM is only generated when the # command is used, or user-defined commands (below) are used. If many such occur in repeated PRINTOUT forms, it may be more efficient to embed them all in a single RESETFORM which changes the primary output file, and then specify *FILE*=NIL in the PRINTOUT expressions themselves.

User-Defined Commands

The collection of commands and options outlined above is aimed at fulfilling all common printing needs. However, certain applications might have other, more specialized printing idioms, so a facility is provided whereby you can define new commands. This is done by adding entries to the global list PRINTOUTMACROS to define how the new commands are to be translated.

PRINTOUTMACROS [Variable]

PRINTOUTMACROS is an association-list whose elements are of the form **(***COMM FN*). Whenever *COMM* appears in command position in the sequence of PRINTOUT commands (as opposed to an argument position of another command), *FN* is applied to the tail of the command-list (including the command).

After inspecting as much of the tail as necessary, the function must return a list whose CAR is the translation of the user-defined command and its arguments, and whose CDR is the list of commands still remaining to be translated in the normal way.

For example, suppose you want to define a command "?", which will cause its single argument to be printed with PRIN1 only if it is not NIL. This can be done by entering (? ?TRAN) on PRINTOUTMACROS, and defining the function ?TRAN as follows:

```
(DEFINEQ (?TRAN (COMS)
   (CONS
      (SUBST (CADR COMS) 'ARG
         '(PROG ((TEMP ARG))
               (COND (TEMP (PRIN1 TEMP)))))
      (CDDR COMS))]
```

Note that ?TRAN does not do any printing itself; it returns a form which, when evaluated in the proper context, will perform the desired action. This form should direct all printing to the primary output file.

Special Printing Functions

The paragraph printing commands are translated into calls on the function PRINTPARA, which may also be called directly:

(**PRINTPARA** *LMARG RMARG LIST P2FLAG PARENFLAG FILE*) [Function]

Prints *LIST* on *FILE* in line-filled paragraph format with its first element beginning at the current line position and ending at or before *RMARG*, and with subsequent lines appearing between *LMARG* and *RMARG*. If *P2FLAG* is non-NIL, prints elements using PRIN2, otherwise PRIN1. If *PARENFLAG* is non-NIL, then parentheses will be printed around the elements of *LIST*.

If *LMARG* is zero or positive, it is interpreted as an absolute column position. If it is negative, then the left margin will be at |*LMARG*|+(POSITION). If *LMARG*=NIL, the left margin will be at (POSITION), and the paragraph will appear in block format.

If *RMARG* is positive, it also is an absolute column position (which may be greater than the current (LINELENGTH)). Otherwise, it is interpreted as relative to (LINELENGTH), i.e., the right margin will be at (LINELENGTH)+|*RMARG*|. Example: (TAB 10) (PRINTPARA 5 -5 LST T) will PRIN2 the elements of LST in a paragraph with the first line beginning at column 10, subsequent lines beginning at column 5, and all lines ending at or before (LINELENGTH)-5.

The current (LINELENGTH) is unaffected by PRINTPARA, and upon completion, *FILE* will be positioned immediately after the last character of the last item of *LIST*. PRINTPARA is a no-op if *LIST* is not a list.

The right-flushing and centering commands translate as calls to the function FLUSHRIGHT:

(**FLUSHRIGHT** *POS X MIN P2FLAG CENTERFLAG FILE*) [Function]

If *CENTERFLAG*=NIL, prints *X* right-flushed against position *POS* on *FILE*; otherwise, centers *X* between the current line position and *POS*. Makes sure that it spaces over at least *MIN* spaces before printing by doing a TERPRI if necessary; *MIN*=NIL is equivalent to *MIN*=1. A positive *POS* indicates an absolute position, while a negative *POS* signifies the position which is |*POS*| to the right of the current line position. *POS*=0 is interpreted as (LINELENGTH), the right margin.

READFILE and WRITEFILE

For those applications where you simply want to simply read all of the expressions on a file, and not evaluate them, the function READFILE is available:

(**READFILE** *FILE RDTBL ENDTOKEN*) [NoSpread Function]

Reads successive expressions from file using READ (with read table *RDTBL*) until the single litatom *ENDTOKEN* is read, or an end of file encountered. Returns a list of these expressions.

If *RDTBL* is not specified, it defaults to FILERDTBL. If *ENDTOKEN* is not specified, it defaults to the litatom STOP.

(**WRITEFILE** *X FILE*) [Function]

Writes a date expression onto *FILE*, followed by successive expressions from *X*, using FILERDTBL as a read table. If *X* is atomic, its value is used. If *FILE* is not open, it is opened. If *FILE* is a list, (CAR *FILE*) is used and the file is left opened. Otherwise, when *X* is finished, the litatom STOP is printed on *FILE* and it is closed. Returns *FILE*.

(**ENDFILE** *FILE*) [Function]

Prints STOP on *FILE* and closes it.

Read Tables

Many Interlisp input functions treat certain characters in special ways. For example, READ recognizes that the right and left parenthesis characters are used to specify list structures, and that the quote character is used to delimit text strings. The Interlisp input and (to a certain extent) output routines are table driven by read tables. Read tables are objects that specify the syntactic properties of characters for input routines. Since the input routines parse character sequences into objects, the read table in use determines which sequences are recognized as literal atoms, strings, list structures, etc.

Most Interlisp input functions take an optional read table argument, which specifies the read table to use when reading an expression. If NIL is given as the read table, the "primary read table" is used. If T is specified, the system terminal read table is used. Some functions will also accept the atom ORIG (*not* the *value* of ORIG) as indicating the "original" system read table. Some output functions also take a read table argument. For example, PRIN2 prints an expression so that it would be read in correctly using a given read table.

The Interlisp-D system uses the following read tables: T for input/output from terminals, the value of FILERDTBL for input/output from files, the value of EDITRDTBL for input from terminals while in the tty-based editor, the value of DEDITRDTBL for input from terminals while in the display-based editor, and the value of CODERDTBL for input/output from compiled files. These five read tables are initially copies of the ORIG read table, with changes made to some of them to provide read macros that are specific to terminal input or file input. Using the functions described below, you may further change, reset, or copy these tables. However, in the case of FILERDTBL and CODERDTBL, you are cautioned that changing these tables may prevent the system from being able to read files made with the original tables, or prevent users possessing only the standard tables from reading files made using the modified tables.

You can also create new read tables, and either explicitly pass them to input/output functions as arguments, or install them as the primary read table, via SETREADTABLE, and then not specify a *RDTBL* argument, i.e., use NIL.

Read Table Functions

(**READTABLEP** *RDTBL*) [Function]

Returns *RDTBL* if *RDTBL* is a real read table (*not* T or ORIG), otherwise NIL.

(**GETREADTABLE** *RDTBL*) [Function]

If *RDTBL*=NIL, returns the primary read table. If *RDTBL*=T, returns the system terminal read table. If *RDTBL* is a real read table, returns *RDTBL*. Otherwise, generates an ILLEGAL READTABLE error.

(**SETREADTABLE** *RDTBL FLG*) [Function]

Sets the primary read table to *RDTBL*. If *FLG*=T, SETREADTABLE sets the system terminal read table, T. Note that you can reset the other system read tables with SETQ, e.g., (SETQ FILERDTBL (GETREADTABLE)).

Generates an ILLEGAL READTABLE error if *RDTBL* is not NIL, T, or a real read table. Returns the previous setting of the primary read table, so SETREADTABLE is suitable for use with RESETFORM (Chapter 14).

(**COPYREADTABLE** *RDTBL*) [Function]

Returns a copy of *RDTBL*. *RDTBL* can be a real read table, NIL, T, or ORIG (in which case COPYREADTABLE returns a copy of the *original* system read table), otherwise COPYREADTABLE generates an ILLEGAL READTABLE error.

Note that COPYREADTABLE is the only function that *creates* a read table.

(**RESETREADTABLE** *RDTBL FROM*) [Function]

Copies (smashes) *FROM* into *RDTBL*. *FROM* and *RDTBL* can be NIL, T, or a real read table. In addition, *FROM* can be ORIG, meaning use the system's original read table.

Syntax Classes

A read table is an object that contains information about the "syntax class" of each character. There are nine basic syntax classes: LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, ESCAPE, BREAKCHAR, SEPRCHAR, and OTHER, each associated with a primitive syntactic property. In addition, there is an unlimited assortment of user-defined syntax classes, known as "read macros". The basic syntax classes are interpreted as follows:

LEFTPAREN (normally left parenthesis) Begins list structure.

RIGHTPAREN (normally right parenthesis) Ends list structure.

LEFTBRACKET (normally left bracket) Begins list structure. Also matches RIGHTBRACKET characters.

RIGHTBRACKET (normally left bracket) Ends list structure. Can close an arbitrary numbers of LEFTPAREN lists, back to the last LEFTBRACKET.

STRINGDELIM (normally double quote) Begins and ends text strings. Within the string, all characters except for the one(s) with class ESCAPE are treated as ordinary, i.e., interpreted as if they were of syntax class OTHER. To include the string delimiter inside a string, prefix it with the ESCAPE character.

ESCAPE (normally percent sign) Inhibits any special interpretation of the next character, i.e., the next character is interpreted to be of class OTHER, independent of its normal syntax class.

BREAKCHAR (None initially) Is a break character, i.e., delimits atoms, but is otherwise an ordinary character.

SEPRCHAR (space, carriage return, etc.) Delimits atoms, and is otherwise ignored.

OTHER Characters that are not otherwise special belong to the class OTHER.

Characters of syntax class LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, and STRINGDELIM are all *break* characters. That is, in addition to their interpretation as delimiting list or string structures, they also terminate the reading of an atom. Characters of class BREAKCHAR serve *only* to terminate atoms, with no other special meaning. In addition, if a break character is the first non-separator encountered by RATOM, it is read as a one-character atom. In order for a break character to be included in an atom, it must be preceded by the ESCAPE character.

Characters of class SEPRCHAR also terminate atoms, but are otherwise completely ignored; they can be thought of as logically spaces. As with break characters, they must be preceded by the ESCAPE character in order to appear in an atom.

For example, if $ were a break character and * a separator character, the input stream ABC**DEF$GH*$$ would be read by six calls to RATOM returning respectively ABC, DEF, $, GH, $, $.

Although normally there is only one character in a read table having each of the list- and string-delimiting syntax classes (such as LEFTPAREN), it is perfectly acceptable for any character to have any syntax class, and for more than one to have the same class.

Note that a "syntax class" is an abstraction: there is no object referencing a collection of characters called a *syntax class*. Instead, a read table provides the *association* between a character and its syntax class, and the input/output routines enforce the abstraction by using read tables to drive the parsing.

The functions below are used to obtain and set the syntax class of a character in a read table. *CH* can either be a character code (a integer), or a character (a single-character atom). Single-digit integers are interpreted as character codes, rather than as characters. For example, 1 indicates Control-A, and 49 indicates the *character* 1. Note that *CH* can be a full sixteen-bit NS character (see Chapter 2).

> Note: Terminal tables, described in Chapter 30, also associate characters with syntax classes, and they can also be manipulated with the functions below. The set of read table and terminal table syntax classes are disjoint, so there is never any ambiguity about which type of table is being referred to.

(**GETSYNTAX** *CH TABLE*) [Function]

Returns the syntax class of *CH*, a character or a character code, with respect to *TABLE*. *TABLE* can be NIL, T, ORIG, or a real read table or terminal table.

CH can also be a syntax class, in which case GETSYNTAX returns a list of the character codes in *TABLE* that have that syntax class.

(**SETSYNTAX** *CHAR CLASS TABLE*) [Function]

Sets the syntax class of *CHAR*, a character or character code, in *TABLE*. *TABLE* can be either NIL, T, or a real read table or terminal table. SETSYNTAX returns the previous syntax class of *CHAR*. *CLASS* can be any one of the following:

- The name of one of the basic syntax classes.
- A list, which is interpreted as a read macro (see below).
- NIL, T, ORIG, or a real read table or terminal table, which means to give *CHAR* the syntax class it has in the table indicated by *CLASS*. For example, (SETSYNTAX '%('ORIG *TABLE*) gives the left parenthesis character in *TABLE* the same syntax class that it has in the original system read table.
- A character code or character, which means to give *CHAR* the same syntax class as the character *CHAR* in *TABLE*. For example, (SETSYNTAX '{ '%[*TABLE*) gives the left brace character the same syntax class as the left bracket.

(**SYNTAXP** *CODE CLASS TABLE*) [Function]

CODE is a character code; *TABLE* is NIL, T, or a real read table or terminal table. Returns T if *CODE* has the syntax class *CLASS* in *TABLE*; NIL otherwise.

CLASS can also be a read macro type (MACRO, SPLICE, INFIX), or a read macro option (FIRST, IMMEDIATE, etc.), in which case SYNTAXP returns T if the syntax class is a read macro with the specified property.

SYNTAXP will *not* accept a character as an argument, only a character *code*.

For convenience in use with SYNTAXP, the atom BREAK may be used to refer to *all* break characters, i.e., it is the union of LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, and BREAKCHAR. For purely symmetrical reasons, the atom SEPR corresponds to all separator characters. However, since the only separator characters are those that also appear in SEPRCHAR, SEPR and SEPRCHAR are equivalent.

Note that GETSYNTAX never returns BREAK or SEPR as a value although SETSYNTAX and SYNTAXP accept them as arguments. Instead, GETSYNTAX returns one of the disjoint basic syntax classes that comprise BREAK. BREAK as an argument to SETSYNTAX is interpreted to mean BREAKCHAR if the character is not already of one of the BREAK classes. Thus, if %(is of class LEFTPAREN, then (SETSYNTAX '%('BREAK) doesn't do anything, since %(is already a break character, but (SETSYNTAX '%('BREAKCHAR) means make %(be *just* a break character, and therefore disables the LEFTPAREN function of %(. Similarly, if one of the format characters is disabled completely, e.g., by (SETSYNTAX '%('OTHER), then (SETSYNTAX '%('BREAK) would make %(be *only* a break character; it would *not* restore %(as LEFTPAREN.

The following functions provide a way of collectively accessing and setting the separator and break characters in a read table:

(**GETSEPR** *RDTBL*) [Function]

Returns a list of separator character codes in *RDTBL*. Equivalent to (GETSYNTAX 'SEPR *RDTBL*).

(**GETBRK** *RDTBL*) [Function]

Returns a list of break character codes in *RDTBL*. Equivalent to (GETSYNTAX 'BREAK *RDTBL*).

(**SETSEPR** *LST FLG RDTBL*) [Function]

Sets or removes the separator characters for *RDTBL*. *LST* is a list of charactors or character codes. *FLG* determines the action of SETSEPR as follows: If *FLG*=NIL, makes *RDTBL* have exactly the elements of *LST* as separators, discarding from *RDTBL* any old separator characters not in *LST*. If *FLG*=0, removes from *RDTBL* as separator characters all elements of *LST*. This provides an "UNSETSEPR". If *FLG*=1, makes each of the characters in *LST* be a separator in *RDTBL*.

If *LST*=T, the separator characters are reset to be those in the system's read table for terminals, regardless of the value of *FLG*, i.e., (SETSEPR T) is equivalent to (SETSEPR (GETSEPR T)). If *RDTBL* is T, then the characters are reset to those in the original system table.

Returns NIL.

(**SETBRK** *LST FLG RDTBL*) [Function]

Sets the break characters for *RDTBL*. Similar to SETSEPR.

As with SETSYNTAX to the BREAK class, if any of the list- or string-delimiting break characters are disabled by an appropriate SETBRK (or by making it be a separator character), its special action for READ will *not* be restored by simply making it be a break character again with SETBRK. However, making these characters be break characters when they already *are* will have no effect.

The action of the ESCAPE character (normally %) is not affected by SETSEPR or SETBRK. It can be disabled by setting its syntax to the class OTHER, and other characters can be used for escape on input by assigning them the class ESCAPE. As of this writing, however, there is no way to change the output escape character; it is "hardwired" as %. That is, on output, characters of special syntax that need to be preceded by the ESCAPE character will always be preceded by %, independent of the syntax of % or which, if any characters, have syntax ESCAPE.

The following function can be used for defeating the action of the ESCAPE character or characters:

(**ESCAPE** *FLG RDTBL*) [Function]

If *FLG*=NIL, makes characters of class ESCAPE behave like characters of class OTHER on input. Normal setting is (ESCAPE T). ESCAPE returns the previous setting.

Read Macros

This is a description of the OLD-INTERLISP-T read macros. Read macros are user-defined syntax classes that can cause complex operations when certain characters are read. Read macro characters are defined by specifying as a syntax class an expression of the form:

(*TYPE OPTION*$_1$ ... *OPTION*$_N$ *FN*)

where *TYPE* is one of MACRO, SPLICE, or INFIX, and *FN* is the name of a function or a lambda expression. Whenever READ encounters a read macro character, it calls the associated function, giving it as arguments the input stream and read table being used for that call to READ. The interpretation of the value returned depends on the type of read macro:

MACRO This is the simplest type of read macro. The result returned from the macro is treated as the expression to be read, instead of the read macro character. Often the macro reads more input itself. For example, in order to cause ~EXPR to be read as (NOT EXPR), one could define ~ as the read macro:

```
[MACRO (LAMBDA (FL RDTBL)
         (LIST 'NOT (READ FL RDTBL]
```

SPLICE The result (which should be a list or NIL) is spliced into the input using NCONC. For example, if $ is defined by the read macro:

```
(SPLICE (LAMBDA NIL (APPEND FOO)))
```

and the value of FOO is (A B C), then when you input (X $ Y), the result will be (X A B C Y).

INFIX The associated function is called with a third argument, which is a list, in TCONC format (Chapter 3), of what has been read at the current level of list nesting. The function's value is taken as a new TCONC list which replaces the old one. For example, the infix operator + could be defined by the read macro:

```
(INFIX (LAMBDA (FL RDTBL Z)
         (RPLACA (CDR Z)
            (LIST (QUOTE IPLUS)
                  (CADR Z)
                  (READ FL RDTBL)))
        Z))
```

If an INFIX read macro character is encountered *not* in a list, the third argument to its associated function is NIL. If the function returns NIL, the read macro character is essentially ignored and reading continues. Otherwise, if the function returns a TCONC list of one element, that element is the value of the READ. If it returns a TCONC list of more than one element, the list is the value of the READ.

The specification for a read macro character can be augmented to specify various options *OPTION*$_1$ ... *OPTION*$_N$, e.g., (MACRO FIRST IMMEDIATE *FN*). The following three disjoint options specify when the read macro character is to be effective:

ALWAYS — The default. The read macro character is always effective (except when preceded by the % character), and is a break character, i.e., a member of (GETSYNTAX 'BREAK *RDTBL*).

FIRST — The character is interpreted as a read macro character *only* when it is the first character seen after a break or separator character; in all other situations, the character is treated as having class OTHER. The read macro character is *not* a break character. For example, the quote character is a FIRST read macro character, so that DON'T is read as the single atom DON'T, rather than as DON followed by (QUOTE T).

ALONE — The read macro character is *not* a break character, and is interpreted as a read macro character only when the character would have been read as a separate atom if it were not a read macro character, i.e., when its immediate neighbors are both break or separator characters.

Making a FIRST or ALONE read macro character be a break character (with SETBRK) disables the read macro interpretation, i.e., converts it to syntax class BREAKCHAR. Making an ALWAYS read macro character be a break character is a no-op.

The following two disjoint options control whether the read macro character is to be protected by the ESCAPE character on output when a litatom containing the character is printed:

ESCQUOTE or ESC — The default. When printed with PRIN2, the read macro character will be preceded by the output escape character (%) as needed to permit the atom containing it to be read correctly. Note that for FIRST macros, this means that the character need be quoted only when it is the first character of the atom.

NOESCQUOTE or NOESC — The read macro character will always be printed without an escape. For example, the **?** read macro in the T read table is a NOESCQUOTE character. Unless you are very careful what you are doing, read macro characters in FILERDTBL should never be NOESCQUOTE, since symbols that happen to contain the read macro character will not read back in correctly.

The following two disjoint options control when the macro's function is actually executed:

IMMEDIATE or IMMED — The read macro character is immediately activated, i.e., the current line is terminated, as if an EOL had been typed, a carriage-return line-feed is printed, and the entire line (including the macro character) is passed to the input function.

IMMEDIATE read macro characters enable you to specify a character that will take effect immediately, as soon as it is encountered in the input, rather than waiting for the line to be terminated. Note that this is not necessarily as soon as the character is *typed*. Characters that cause action as soon as they are typed are interrupt characters (see Chapter 30).

Note that since an IMMEDIATE macro causes any input before it to be sent to the reader, characters typed before an IMMEDIATE read macro character cannot be erased by Control-A or Control-Q once the IMMEDIATE character has been typed, since they have already passed through the line buffer. However, an INFIX read macro can still alter some of what has been typed earlier, via its third argument.

NONIMMEDIATE or NONIMMED The default. The read macro character is a normal character with respect to the line buffering, and so will not be activated until a carriage-return or matching right parenthesis or bracket is seen.

Making a read macro character be both ALONE and IMMEDIATE is a contradiction, since ALONE requires that the next character be input in order to see if it is a break or separator character. Thus, ALONE read macros are always NONIMMEDIATE, regardless of whether or not IMMEDIATE is specified.

Read macro characters can be "nested". For example, if = is defined by

```
(MACRO (LAMBDA (FL RDTBL)
    (EVAL (READ FL RDTBL))))
```

and ! is defined by

```
(SPLICE (LAMBDA (FL RDTBL)
    (READ FL RDTBL)))
```

then if the value of FOO is (A B C), and (X =FOO Y) is input, (X (A B C) Y) will be returned. If (X !=FOO Y) is input, (X A B C Y) will be returned.

Note: If a read macro's function calls READ, and the READ returns NIL, the function cannot distinguish the case where a RIGHTPAREN or RIGHTBRACKET followed the read macro character, (e.g. "(A B ')"), from the case where the atom NIL (or "()") actually appeared. In Interlisp-D, a READ inside of a read macro when the next input character is a RIGHTPAREN or RIGHTBRACKET reads the character and returns NIL, just as if the READ had not occurred inside a read macro.

If a call to READ from within a read macro encounters an unmatched RIGHTBRACKET *within* a list, the bracket is simply put back into the buffer to be read (again) at the higher level. Thus, inputting an expression such as (A B '(C D] works correctly.

(**INREADMACROP**) [Function]

Returns NIL if currently *not* under a read macro function, otherwise the number of unmatched left parentheses or brackets.

(**READMACROS** *FLG RDTBL*) [Function]

If *FLG*=NIL, turns off action of read macros in read table *RDTBL*. If *FLG*=T, turns them on. Returns previous setting.

The following read macros are standardly defined in Interlisp in the T and EDITRDTBL read tables:

' (single-quote) Returns the next expression, wrapped in a call to QUOTE; e.g., 'FOO reads as (QUOTE FOO). The macro is defined as a FIRST read macro, so that the quote character has no effect in the middle of a symbol. The macro is also ignored if the quote character is immediately followed by a separator character.

Control-Y Defined in T and EDITRDTBL. Returns the result of evaluating the next expression. For example, if the value of FOO is (A B), then (LIST 1 *control-Y*FOO 2) is read as (LIST 1 (A B) 2). Note that no structure is copied; the third element of that input expression is still EQ to the value of FOO. Control-Y can thus be used to read structures that ordinarily have no read syntax. For example, the value returned from reading (KEY1 *Control-Y*(ARRAY 10)) has an array as its second element. Control-Y can be thought of as an "un-quote" character. The choice of character to perform this function is changeable with SETTERMCHARS (see Chapter 16).

` (backquote) Backquote makes it easier to write programs to construct complex data structures. Backquote is like quote, except that within the backquoted expression, forms can be evaluated. The general idea is that the backquoted expression is a "template" containing some constant parts (as with a quoted form) and some parts to be filled in by evaluating something. Unlike with control-Y, however, the evaluation occurs not at the time the form is read, but at the time the backquoted expression is evaluated. That is, the backquote macro returns an expression which, when evaluated, produces the desired structure.

Within the backquoted expression, the character "," (comma) introduces a form to be evaluated. The value of a form preceded by ",@" is to be spliced in, using APPEND. If it is permissible to destroy the list being spliced in (i.e., NCONC may be used in the translation), then ",." can be used instead of ",@".

For example, if the value of FOO is (1 2 3 4), then the form

```
`(A ,(CAR FOO) ,@(CDDR FOO) D E)
```

evaluates to `(A 1 3 4 D E)`; it is logically equivalent to writing

```
(CONS 'A
   (CONS (CAR FOO)
      (APPEND (CDDR FOO) '(D E))))
.
```

Backquote is particularly useful for writing macros. For example, the body of a macro that refers to `X` as the macro's argument list might be

```
`(COND
   ((FIXP ,(CAR X))
      ,(CADR X))
   (T .,(CDDR X)))
```

which is equivalent to writing

```
(LIST 'COND
     (LIST (LIST 'FIXP (CAR X))
           (CADR X))
     (CONS 'T (CDDR X)))
```

Note that comma does *not* have any special meaning outside of a backquote context.

For users without a backquote character on their keyboards, backquote can also be written as `|'` (vertical-bar, quote).

`?` Implements the `?=` command for on-line help regarding the function currently being "called" in the typein (see Chapter 26).

| (vertical bar) When followed by an end of line, tab or space, | is ignored, i.e., treated as a separator character, enabling the editor's `CHANGECHAR` feature (see Chapter 26). Otherwise it is a "dispatching" read macro whose meaning depends on the character(s) following it. The following are currently defined:

`'` (quote) -- A synonym for backquote.

`.` (period) -- Returns the evaluation of the next expression, i.e., this is a synonym for Control-Y.

, (comma) -- Returns the evaluation of the next expression *at load time*, i.e., the following expression is quoted in such a manner that the compiler treats it as a literal whose value is not determined until the compiled expression is loaded.

`O` or `o` (the letter O) -- Treats the next number as octal, i.e., reads it in radix 8. For example, `|o12` = `10` (decimal).

`B` or `b` -- Treats the next number as binary, i.e., reads it in radix 2. For example, `|b101` = `5` (decimal).

`X` or `x` -- Treats the next number as hexadecimal, i.e., reads it in radix 16. The uppercase letters `A` though `F` are used as the digits after `9`. For example, `|x1A = 26` (decimal).

`R` or `r` -- Reads the next number in the radix specified by the (decimal) number that appears between the `|` and the `R`. When inputting a number in a radix above ten, the upper-case letters `A` through `Z` can be used as the digits after `9` (but there is no digit above `Z`, so it is not possible to type all base-99 digits). For example, `|3r120` reads 120 in radix 3, returning 15.

`(`, `{`, `^` -- Used internally by `HPRINT` and `HREAD` (see above) to print and read unusual expressions.

The dispatching characters that are letters can appear in either upper- or lowercase.

25. USER/ INPUT/OUTPUT PACKAGES

Interlisp-D can perform input/output operations on a large variety of physical devices.

This chapter presents a number of packages that have been developed for displaying and allowing the user to enter information. These packages are used to implement the user interface of many system facilities.

INSPECT (see the INSPECT section below) provides a window-based facility for displaying and changing the fields of a data object.

PROMPTFORWORD (see the PROMPTFORWORD section below) is a function used for entering a simple string of characters. Basic editing and prompting facilities are provided.

ASKUSER (see the ASKUSER section below) provides a more complicated prompting and answering facility, allowing a series of questions to be printed. Prompts and argument completion are supported.

TTYIN (see the TTYIN Display Typein Editor section below) is a display typein editor, that provides complex text editing facilities when entering an input line.

PRETTYPRINT (see the Prettyprint section below) is used for printing function definitions and other list structures, using multiple fonts and indenting lines to show the structure of the list.

Inspector

The Inspector provides a display-oriented facility for looking at and changing arbitrary Interlisp-D data structures. The inspector can be used to inspect all user datatypes and many system datatypes (although some objects such as numbers have no inspectable structure). The inspector displays the field names and values of an arbitrary object in a window that allows setting of the properties and further inspection of the values. This latter feature makes it possible to "walk" around all of the data structures in the system at the touch of a button. In addition, the inspector is integrated with the break package to allow inspection of any object on the stack and with the display and teletype structural editors to allow the editors to be used to "inspect" list structures and the inspector to "edit" datatypes.

The underlying mechanisms of the data inspector have been designed to allow their use as specialized editors in user applications. This functionality is described at the end of this section.

Note: Currently, the inspector does not have UNDOing. Also, variables whose values are changed will not be marked as such.

Calling the Inspector

There are several ways to open an inspect window onto an object. In addition to calling `INSPECT` directly (below), the inspector can also be called by buttoning an Inspect command inside an existing inspector window. Finally, if a non-list is edited with `EDITDEF` (see Chapter 17), the inspector is called. This also causes the inspector to be called by the Dedit command from the display editor or the `EV` command from the teletype editor if the selected piece of structure is a non-list.

(**INSPECT** *OBJECT ASTYPE WHERE*) [Function]

Creates an inspect window onto *OBJECT*. If *ASTYPE* is given, it will be taken as the record type of *OBJECT*. This allows records to be inspected with their property names. If *ASTYPE* is `NIL`, the data type of *OBJECT* will be used to determine its property names in the inspect window.

WHERE specifies the location of the inspect window. If *WHERE* is `NIL`, the user will be prompted for a location. If *WHERE* is a window, it will be used as the inspect window. If *WHERE* is a region, the inspect window will be created in that region of the screen. If *WHERE* is a position, the inspect window will have its lower left corner at that position on the screen.

`INSPECT` returns the inspect window onto *OBJECT*, or `NIL` if no inspection took place.

(**INSPECTCODE** *FN WHERE* — — — —) [Function]

Opens a window and displays the compiled code of the function *FN* using `PRINTCODE`. The window is scrollable.

`WHERE` determines where the window should appear. It can be a position, a region, or a window. If `NIL`, the user is prompted to specify the position of the window.

Note: If the Tedit library package is loaded, `INSPECTCODE` uses it to create the code inspector window. Also, if `INSPECTCODE` is called to inspect the frame name in a break window (see Chapter 14), the location in the code that the frame's PC indicates it was executing at the time is highlighted.

Multiple Ways of Inspecting

For some datatypes there is more than one aspect that is of interest or more than one method of inspecting the object. In these cases, the inspector will bring up a menu of the possibilities and wait for the user to select one.

If the object is a litatom, the commands are the types for which the litatom has definitions as determined by `HASDEF`. Some typical commands are:

`FNS` Edit the definition of the selected litatom.

■ `VARS` Inspect the value.

`PROPS` Inspect the property list.

If the object is a list, there will be choice of how to inspect the list:

`Inspect`	Opens an inspect window in which the properties are numbers and the values are the elements of the list.
`TtyEdit`	Calls the teletype list structure editor on the list (see Chapter 16).
`DisplayEdit`	Calls the DEdit display editor on the list (see Chapter 16).
As a `PLIST`	Inspects the list as a property list, if the list is in property list form: `((PROP`$_1$ `VAL`$_1$`) ... (PROP`$_N$ `VAL`$_N$`))`.
As an `ALIST`	Inspects the list as an association-list, if the list is in `ASSOC` list form: `(PROP`$_1$ `VAL`$_1$ `... PROP`$_N$ `VAL`$_N$`)`.
As a record	Brings up a submenu with all of the `RECORD`s in the system and inspect the list with the one chosen.
As a "record type"	Inspects the list as the record of the type named in its `CAR`, if the `CAR` of the list is the name of a `TYPERECORD` (see Chapter 8).

If the object is a bitmap, the choice is between inspecting the bitmap's contents with the bitmap editor (`EDITBM`) or inspecting the bitmap's fields.

Other datatypes may include multiple methods for inspecting objects of that type.

Inspect Windows

An inspect window displays two columns of values. The lefthand column lists the property names of the structure being inspected. The righthand column contains the values of the properties named on the left. For variable length data such as lists and arrays, the "property names" are numbers from 1 to the length of the inspected item and the values are the corresponding elements. For arrays, the property names are the array element numbers and the values are the corresponding elements of the array.

For large lists or arrays, or datatypes with many fields, the initial window may be too small to contain all of them. In these cases, the unseen elements can be scrolled into view (from the bottom) or the window can be reshaped to increase its size.

In an inspect window, the `LEFT` button is used to select things, the `MIDDLE` button to invoke commands that apply to the selected item. Any property or value can be selected by pointing the cursor directly at the text representing it, and clicking the `LEFT` button. There is one selected item per window and it is marked by having its surrounding box inverted.

The options offered by the `MIDDLE` button depend on whether the selection is a property or a value. If the selected item is a value, the options provide different ways of inspecting the selected structure. The exact commands that are given depend on the type of the value. An example of the menu you may see is:

If the selected item is a property name, the command SET will appear. If selected, the user will be asked to type in an expression, and the selected property will be set to the result of evaluating the read form. The evaluation of the read form and the replacement of the selected item property will appear as their own history events and are individually undoable. Properties of system datatypes cannot be set. (There are often consistency requirements which can be inadvertently violated in ways that crash the system. This may be true of some user datatypes as well, however the system doesn't know which ones. Users are advised to exercise caution.)

It is possible to copy-select property names or values out of an inspect window. Litatoms, numbers and strings are copied as they are displayed. Unprintable objects (such as bitmaps, etc.) come out as an appropriate system expression, such that if is evaluated, the object is re-created.

Inspect Window Commands

By pressing the MIDDLE button in the title of the inspect window, a menu of commands that apply to the inspect window is brought up:

ReFetch [Inspect Window Command]

An inspect window is not automatically updated when the structure it is inspecting is changed. The ReFetch command will refetch and redisplay all of the fields of the object being inspected in the inspect window.

IT←datum [Inspect Window Command]

Sets the variable IT to object being inspected in the inspect window.

IT←selection [Inspect Window Command]

Sets the variable IT to the property name or value currently selected in the inspect window.

Interaction With Break Windows

The break window facility (see Chapter 14) knows about the inspector in the sense that the backtrace frame window is an inspect window onto the frame selected from the back trace menu during a break. Thus you can call the inspector on an object that is bound on the stack by selecting its frame in the back trace menu, selecting its value with the LEFT button in the back trace frame window, and selecting the inspect command with the MIDDLE button in the back trace frame window. The values of variables in frames can be set by selecting the variable name with the LEFT button and then the "Set" command with the MIDDLE button.

Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. Exercise caution in setting variables in other than your own code.

Controlling the Amount Displayed During Inspection

The amount of information displayed during inspection can be controlled using the following variables:

MAXINSPECTCDRLEVEL [Variable]

The inspector prints only the first MAXINSPECTCDRLEVEL elements of a long list, and will make the tail containing the unprinted elements the last item. The last item can be inspected to see further elements. Initially 50.

MAXINSPECTARRAYLEVEL [Variable]

The inspector prints only the first MAXINSPECTARRAYLEVEL elements of an array. The remaining elements can be inspected by calling the function (INSPECT/ARRAY ARRAY BEGINOFFSET) which inspects the BEGINOFFSET through the BEGINOFFSET + MAXINSPECTARRAYLEVEL elements of ARRAY. Initially 300.

INSPECTPRINTLEVEL [Variable]

When printing the values, the inspector resets PRINTLEVEL (see Chapter 25) to the value of INSPECTPRINTLEVEL. Initially (2 . 5).

INSPECTALLFIELDSFLG [Variable]

If INSPECTALLFIELDSFLG is T, the inspector will show computed fields (ACCESSFNS, Chapter 8) as well as regular fields for structures that have a record definition. Initially T.

Inspect Macros

The Inspector can be extended to inspect new structures and datatypes by adding entries to the list INSPECTMACROS. An entry should be of the form (OBJECTTYPE . INSPECTINFO). OBJECTTYPE is used to determine the types of objects that are inspected with this macro. If OBJECTTYPE is a litatom, the INSPECTINFO will be used to inspect items whose type name is OBJECTTYPE. If OBJECTTYPE is a list of the form (FUNCTION DATUM-PREDICATE), DATUM-PREDICATE will be APPLYed to the item and if it returns non-NIL, the INSPECTINFO will be used to inspect the item.

INSPECTINFO can be one of two forms. If INSPECTINFO is a litatom, it should be a function that will be applied to three arguments (the item being inspected, OBJECTTYPE, and the value of WHERE passed to INSPECT) that should do the inspection. If INSPECTINFO is not a litatom, it should be a list of (PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN) where the elements of this list are the arguments for INSPECTW.CREATE, described below. From this list, the WHERE argument will be evaluated; the others will not. If WHERE is NIL, the value of WHERE that was passed to INSPECT will be used.

Examples:

The entry ((FUNCTION MYATOMP) PROPNAMES GETPROP PUTPROP) on INSPECTMACROS would cause all objects satisfying the predicate MYATOMP to have their properties inspected with GETPROP and PUTPROP. In this example, MYATOMP should make sure the object is a litatom.

The entry (MYDATATYPE . MYINSPECTFN) on INSPECTMACROS would cause all datatypes of type MYDATATYPE to be passed to the function MYINSPECTFN.

INSPECTWs

The inspector is built on the abstraction of an INSPECTW. An INSPECTW is a window with certain window properties that display an object and respond to selections of the object's parts. It is characterized by an object and its list of properties. An INSPECTW displays the object in two columns with the property names on the left and the values of those properties on the right. An INSPECTW supports the protocol that the LEFT mouse button can be used to select any property name or property value and the MIDDLE button calls a user provided function on the selected value or property. For the Inspector application, this function puts up a menu of the alternative ways of inspecting values or of the ways of setting properties. INSPECTWs are created with the following function:

(**INSPECTW.CREATE** *DATUM PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN*) [Function]

Creates an INSPECTW that views the object *DATUM*. If *PROPERTIES* is a list, it is taken as the list of properties of *DATUM* to display. If *PROPERTIES* is a litatom, it is APPLYed to *DATUM* and the result is used as the list of properties to display.

FETCHFN is a function of two arguments (OBJECT PROPERTY) that should return the value of the PROPERTY property of OBJECT. The result of this function will be printed (with PRIN2) in the INSPECTW as the value.

STOREFN is a function of three arguments (OBJECT PROPERTY NEWVALUE) that changes the PROPERTY property of OBJECT to NEWVALUE. It is used by the default *PROPCOMMANDFN* and *VALUECOMMANDFN* to change the value of a property and also by the function INSPECTW.REPLACE (described below). This can be NIL if the user provides command functions which do not call INSPECTW.REPLACE. Each replace action will be a separate event on the history list. Users are encouraged to provide UNDOable STOREFNs.

PROPCOMMANDFN is a function of three arguments (PROPERTY OBJECT INSPECTW) which gets called when the user presses the MIDDLE button and the selected item in the

INSPECTW is a property name. PROPERTY will be the name of the selected property, OBJECT will be the datum being viewed, and INSPECTW will be the window. If *PROPCOMMANDFN* is a string, it will get printed in the PROMPTWINDOW when the MIDDLE button is pressed. This provides a convenient way to notify the user about disabled commands on the properties. DEFAULT.INSPECTW.PROPCOMMANDFN, the default *PROPCOMMANDFN*, will present a menu with the single command Set on it. If selected, the Set command will read a value from the user and set the selected property to the result of EVALuating this read value.

VALUECOMMANDFN is a function of four arguments (VALUE PROPERTY OBJECT INSPECTW) that gets called when the user presses the MIDDLE button and the selected item in the INSPECTW is a property value. VALUE will be the selected value (as returned by *FETCHFN*), PROPERTY will be the name of the property VALUE is the value of, OBJECT will be the datum being viewed, and INSPECTW will be the INSPECTW window.

DEFAULT.INSPECTW.VALUECOMMANDFN, the default *VALUECOMMANDFN*, will present a menu of possible ways of inspecting the value and create a new Inspect window if one of the menu items is selected.

TITLECOMMANDFN is a function of two arguments (INSPECTW OBJECT) which gets called when the user presses the MIDDLE button and the cursor is in the title or border of the inspect window INSPECTW. This command function is provided so that users can implement commands that apply to the entire object. The default *TITLECOMMANDFN* (DEFAULT.INSPECTW.TITLECOMMANDFN) presents a menu with the commands ReFetch, IT←datum, and IT←election .

TITLE specifies the title of the window. If *TITLE* is NIL, the title of the window will be the printed form of *DATUM* followed by the string " Inspector". If *TITLE* is the litatom DON'T, the inspect window will not have a title. If *TITLE* is any other litatom, it will be applyed to the *DATUM* and the potential inspect window (if it is known). If this result is the litatom DON'T, the inspect window will not have a title; otherwise the result will be used as a title. If *TITLE* is not a litatom, it will be used as the title.

SELECTIONFN is a function of three arguments (PROPERTY VALUEFLG INSPECTW) which gets called when the user releases the left button and the cursor is on one of the items. The *SELECTIONFN* allows a program to take action on the user's selection of an item in the inspect window. At the time this function is called, the selected item has been "selected". The function INSPECTW.SELECTITEM (described below) can be used to turn off this selection. PROPERTY will be the name of the property of the selected item. *VALUEFLG* will be NIL if the selected item is the property name; T if the selected item is the property value.

WHERE indicates where the inspect window should go. Its interpretation is described in INSPECT (see above).

PROPPRINTFN is a function of two arguments (PROPERTY DATUM) which gets called to determine what to print in the property place for the property PROPERTY. If *PROPPRINTFN* returns NIL, no property name will be printed and the value will be printed to the left of the other values.

An inspect window uses the following window property names to hold information: *DATUM*, *FETCHFN*, *STOREFN*, *PROPCOMMANDFN*, *VALUECOMMANDFN*, *SELECTIONFN*, *PROPPRINTFN*, *INSPECTWTITLE*, *PROPERTIES*, *CURRENTITEM* and *SELECTABLEITEMS*.

(**INSPECTW.REDISPLAY** *INSPECTW PROPS* —) [Function]

Updates the display of the objects being inspected in INSPECTW. If *PROPS* is a property name or a list of property names, only those properties are updated. If *PROPS* is NIL, all properties are redisplayed. This function is provided because inspect windows do not automatically update their display when the object they are showing changes.

This function is called by the ReFetch command in the title command menu of an INSPECTW (see above).

(**INSPECTW.REPLACE** *INSPECTW PROPERTY NEWVALUE*) [Function]

Calls the STOREFN of the inspect window *INSPECTW* to change the property named *PROPERTY* to the value *NEWVALUE* and updates the display of *PROPERTY*'s value in the display. This provides a functional interface for user PROPCOMMANDFNs.

(**INSPECTW.SELECTITEM** *INSPECTW PROPERTY VALUEFLG*) [Function]

Sets the selected item in an inspect window. The item is inverted on the display and put on the window property CURRENTITEM of *INSPECTW*. If *INSPECTW* has a CURRENTITEM, it is deselected. *PROPERTY* is the name of the property of the selected item. *VALUEFLG* is NIL if the selected item is the property name; T if the selected item is the property value. If *PROPERTY* is NIL, no item will be selected. This provides a way of deselecting all items.

PROMPTFORWORD

PROMPTFORWORD is a function that reads in a sequence of characters, generally from the keyboard, without involving READ-like syntax. A user can supply a prompting string, as well as a "candidate" string, which is printed and used if the user types only a word terminator character (or doesn't type anything before a given time limit). As soon as any characters are typed the "candidate" string is erased and the new input takes its place.

PROMPTFORWORD accepts user type-in until one of the "word terminator" characters is typed. Normally, the word terminator characters are EOL, ESCAPE, LF, SPACE, or TAB. This list can be changed using the TERMINCHAR.LST argument to PROMPTFORWORD, for example if it is desirable to allow the user to input lines including spaces.

PROMPTFORWORD also recognizes the following special characters:

Control-A
BACKSPACE
DELETE Any of these characters deletes the last character typed and appropriately erases it from the echo stream if it is a display stream.

Control-Q Erases all the type-in so far.

Control-R Reprints the accumulated string.

Control-V "Quotes" the next character: after typing Control-V, the next character typed is added to the accumulated string, regardless of any special meaning it has. Allows the user to include editing characters and word terminator characters in the accumulated string.

Control-W Erases the last word.

? Calls up a "help" facility. The action taken is defined by the GENERATE?LIST.FN argument to PROMPTFORWORD (see below). Normally, this prints a list of possible candidates.

(**PROMPTFORWORD** *PROMPT.STR CANDIDATE.STR GENERATE?LIST.FN ECHO.CHANNEL DONTECHOTYPEIN.FLG URGENCY.OPTION TERMINCHARS.LST KEYBD.CHANNEL*) [Function]

PROMPTFORWORD has a multiplicity of features, which are specified through a rather large number of input arguments, but the default settings for them (i.e., when they aren't given, or are given as NIL) is such to minimize the number needed in the average case, and an attempt has been made to order the more frequently non-defaulted arguments at the beginning of the argument list. The default input and echo are both to the terminal; the terminal table in effect during input allows most control characters to be INDICATE'd.

PROMPTFORWORD returns NIL if a null string is typed; this would occur when no candidate is given and only a terminator is typed, or when the candidate is erased and a terminator is typed with no other input still un-erased. In all other cases, PROMPTFORWORD returns a string.

PROMPTFORWORD is controlled through the following arguments:

PROMPT.STR If non-NIL, this is coerced to a string and used for prompting; an additional space is output after this string.

CANDIDATE.STR If non-NIL, this is coerced to a string and offered as initial contents of the input buffer.

GENERATE?LIST.FN If non-NIL, this is either a string to be printed out for help, or a function to be applied to *PROMPT.STR* and *CANDIDATE.STR* (after both have been coerced to strings), and which should return a list of potential candidates. The help string or list of potential candidates will then be printed on a separate line, the prompt will be restarted, and any type-in will be re-echoed.
Note: If *GENERATE?LIST.FN* is a function, its value list will be cached so that it will be run at most once per call to PROMPTFORWORD.

ECHO.CHANNEL Coerced to an output stream; NIL defaults to T, the "terminal output stream", normally (TTYDISPLAYSTREAM). To achieve echoing to the "current output stream", use (GETSTREAM NIL 'OUTPUT). If echo is to a display stream, it will have a flashing caret showing where the next input is to be echoed.

DONTECHOTYPEIN.FLG If T, there is no echoing of the input characters. If the value of *DONTECHOTYPEIN.FLG* is a single-character atom or string, that character is echoed instead of the actual input. For example, LOGIN prompts for a password with *DONTECHOTYPEIN.FLG* being "*".

URGENCY.OPTION If NIL, PROMPTFORWORD quietly wait for input, as READ does; if a number, this is the number of seconds to wait for the user to respond (if timeout is reached, then *CANDIDATE.WORD* is returned, regardless of any other type-in activity); if T, this means to wait forever, but periodically flash the window to alert the user; if TTY, then PROMPTFORWORD grabs the TTY immediately. When URGENCY.OPTION = TTY, the cursor is temporarily changed to a different shape to indicate the urgent nature of the request.

TERMINCHARS.LST This is list of "word terminator" character codes; it defaults to (CHARCODE (EOL ESCAPE LF SPACE TAB)). This may also be a single character code.

KEYBD.CHANNEL If non-NIL, this is coerced to a stream, and the input bytes are taken from that stream. NIL defaults to the keyboard input stream. Note that this is not the same as the terminal input stream T, which is a buffered keyboard input stream, not suitable for use with PROMPTFORWORD.

Examples:

```
(PROMPTFORWORD
   "What is your FOO word?" 'Mumble
   (FUNCTION (LAMBDA () '(Grumble Bletch)))
   PROMPTWINDOW NIL 30)
```

This first prompts the user for input by printing the first argument as a prompt into PROMPTWINDOW; then the proffered default answer, Mumble, is printed out and the caret starts flashing just after it to indicate that the upcoming input will be echoed there. If the user fails to complete a word within 30 seconds, then the result will be the string Mumble.

```
(FRESHLINE T)
(LIST
   (PROMPTFORWORD
      (CONCAT "{" HOST "} Login:")
      (USERNAME NIL NIL T))
   (PROMPTFORWORD
      " (password)" NIL NIL NIL '*))
```

This first prompts in whatever window is currently (TTYDISPLAYSTREAM), and then takes in a username; the second call prompts with (password) and takes in another word (the password) without proffering a candidate, echoing the typed-in characters as "*".

ASKUSER

DWIM, the compiler, the editor, and many other system packages all use ASKUSER, an extremely general user interaction package, for their interactions with the user at the terminal. ASKUSER takes as its principal argument KEYLST which is used to drive the interaction. KEYLST specifies what the user can type at any given point, how ASKUSER should respond to the various inputs, what value should be returned by ASKUSER, and is also used to present the user at any given point with a list of the possible responses. ASKUSER also takes other arguments which permit specifying a wait time, a default value, a message to be printed on entry, a flag indicating whether or not typeahead is to be permitted, a flag indicating whether the transaction is to be stored on the history list (see Chapter 13), a default set of options, and an (optional) input file / string.

(**ASKUSER** *WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRNTFLG OPTIONSLST FILE*) [Function]

WAIT is either NIL or a number (of seconds). *DEFAULT* is a single character or a sequence (list) of characters to be used as the default inputs for the case when *WAIT* is not NIL and more than *WAIT* seconds elapse without any input. In this case, the character(s) from *DEFAULT* are processed exactly as though they had been typed, except that ASKUSER first types ". . .".

MESS is the initial message to be printed by ASKUSER, if any, and can be a string, or a list. In the latter case, each element of the list is printed, separated by spaces, and terminated with a " ? ". *KEYLST* and *OPTIONSLST* are described. *TYPEAHEAD* is T if the user is permitted to typeahead a response to ASKUSER. NIL means any typeahead should be cleared and saved. *LISPXPRNTFLG* determines whether or not the interaction is to be recorded on the history list. *FILE* can be either NIL (in which case it defaults to the terminal input stream, T) or a stream.

All input operations take place from *FILE* until an unacceptable input is encountered, i.e., one that does not conform to the protocol defined by *KEYLST*. At that point, *FILE* is set to T, *DEFAULT* is set to NIL, the input buffer is cleared, and a bell is rung. Unacceptable inputs are not echoed.

The value of ASKUSER is the result of packing all the keys that were matched, unless the *RETURN* option is specified (see the Options section below).

(**MAKEKEYLST** *LST DEFAULTKEY LCASEFLG AUTOCOMPLETEFLG*) [Function]

LST is a list of atoms or strings. MAKEKEYLST returns an ASKUSER KEYLST which will permit the user to specify one of the elements on *LST* by either typing enough characters to make the choice unambiguous, or else typing a number between 1 and N, where N is the length of *LST*.

For example, if ASKUSER is called with KEYLST = (MAKEKEYLST '(CONNECT SUPPORT COMPILE)), then the user can type C-O-N, S, C-O-M, 1, 2, or 3 to indicate one of the three choices.

If *LCASEFLG* = T, then echoing of upper case elements will be in lower case (but the value returned will still be one of the elements of LST). If *DEFAULTKEY* is non-NIL, it will be the last key on the KEYLST. Otherwise, a key which permits the user to indicate "No - none of the above" choices, in which case the value returned by ASKUSER will be NIL.

AUTOCOMPLETEFLG is used as the value of the *AUTOCOMPLETEFLG* option of the resulting key list.

Format of KEYLST

KEYLST is a list of elements of the form (KEY PROMPTSTRING . OPTIONS), where KEY is an atom or a string (equivalent), PROMPTSTRING is an atom or a string, and OPTIONS a list of options in property list format. The options are explained below. If an option is specified in OPTIONS, the value of the option is the next element. Otherwise, if the option is specified in the OPTIONSLST argument to ASKUSER, its value is the next element on OPTIONSLST. Thus, OPTIONSLST can be used to provide default options for an entire KEYLST, rather than having to include the option at each level. If an option does not appear on either OPTIONS or OPTIONSLST, its value is NIL.

For convenience, an entry on KEYLST of the form (KEY . ATOM/STRING), can be used as an abbreviation for (KEY ATOM/STRING CONFIRMFLG T), and an entry of just the form KEY, i.e., a non-list, as an abbreviation for (KEY NIL CONFIRMFLG T).

As each character is read, it is matched against the currently active keys. A character matches a key if it is the same character as that in the corresponding position in the key, or, if the character is an alphabetic character, if the characters are the same without regard for upper/lower case differences, i.e. "A" matches "a" and vice versa (unless the NOCASEFLG option is T, see the Options section below). In other words, if two characters have already been input and matched, the third character is matched with each active key by comparing it with the third character of that key. If the character matches with one or more of the keys, the entries on KEYLST corresponding to the remaining keys are discarded. If the character does not match with any of the keys, the character is not echoed, and a bell is rung instead.

When a key is complete, PROMPTSTRING is printed (NIL is equivalent to "", the empty string, i.e., nothing will be printed). Then, if the value of the CONFIRMFLG option is T, ASKUSER waits for confirmation of the key by a carriage return or space. Otherwise, the key does not require confirmation.

Then, if the value of the KEYLST option is not NIL, its value becomes the new KEYLST, and the process recurses. Otherwise, the key is a "leaf," i.e., it terminates a particular path through the original, top-level KEYLST, and ASKUSER returns the result of packing all the keys that have been matched and completed along the way (unless the RETURN option is used to specify some other value, as described below).

For example, when ASKUSER is called with KEYLST = NIL, the following KEYLST is used as the default:

```
((Y "escr") (N "ocr"))
```

This KEYLST specifies that if (as soon as) the user types Y (or y), ASKUSER echoes with Y, prompts with escr, and returns Y as its value. Similarly, if the user types N, ASKUSER echoes the N, prompts with ocr, and returns N. If the user types ?, ASKUSER prints:

```
Yes
No
```

to indicate his possible responses. All other inputs are unacceptable, and ASKUSER will ring the bell and not echo or print anything.

For a more complicated example, the following is the KEYLST used for the compiler questions:

```
((ST "ore and redefine " KEYLST ("" (F . "orget
exprs"))
 (S . "ame as last time")
 (F . "File only")
 (T . "o terminal")
 1
 2
 (Y . "es")
 (N . "o"))
```

When ASKUSER is called with this KEYLST, and the user types an S, two keys are matched: ST and S. The user can then type a T, which matches only the ST key, or confirm the S key by typing a cr or space. If the user confirms the S key, ASKUSER prompts with "ame as last time", and returns S as its value. (Note that the confirming character is not included in the value.) If the user types a T, ASKUSER prompts with "ore and redefine", and makes ("" (F . "orget exprs")) be the new KEYLST, and waits for more input. The user can then type an F, or confirm the "" (which essentially starts out with all of its characters matched). If he confirms the "", ASKUSER returns ST as its value the result of packing ST and "". If he types F, ASKUSER prompts with "orget exprs", and waits for confirmation again. If the user then confirms, ASKUSER returns STF, the result of packing ST and F.

At any point the user can type a ? and be prompted with the possible responses. For example, if the user types S and then ?, ASKUSER will type:

```
STore and redefine Forget exprs
STore and redefine
Same as last time
```

Options

KEYLST — When a key is complete, if the value of the KEYLST option is not NIL, this value becomes the new KEYLST and the process recurses. Otherwise, the key terminates a path through the original, top-level KEYLST, and ASKUSER returns the indicated value.

CONFIRMFLG — If T, the key must be confirmed with either a carriage return or a space. If the value of CONFIRMFLG is a list, the confirming character may be any member of the list.

PROMPTCONFIRMFLG — If T, whenever confirmation is required, the user is prompted with the string [confirm].

NOCASEFLG — If T, says do not perform case independent matching on alphabetic characters. If NIL, do perform case independent matching, i.e. "A" matches with "a" and vice versa.

RETURN — If non-NIL, EVAL of the value of the RETURN option is returned as the value of ASKUSER. Note that different RETURN options can be specified for different keys. The variable ANSWER is bound in ASKUSER to the list of keys that have been matched. In other words, RETURN (PACK ANSWER) would be equivalent to what ASKUSER normally does.

NOECHOFLG — If non-NIL, characters that are matched (or automatically supplied as a result of typing $ (escape) or confirming) are not echoed, nor is the confirming character, if any. The value of NOECHOFLG is automatically NIL when ASKUSER is reading from a file or string. The decision about whether or not to echo a character that matches several keys is determined by the value of the NOECHOFLG option for the first key.

EXPLAINSTRING — If the value of the EXPLAINSTRING option is non-NIL, its value is printed when the user types a ?, rather than KEY + PROMPTSTRING. EXPLAINSTRING enables more elaborate explanations in response to a ? than what the user sees when he is prompted as a result of simply completing keys. For example: One of the entries on the KEYLST used by ADDTOFILES? is:

```
(] "Nowherecr" NOECHOFLG T
 EXPLAINSTRING "] - nowhere, item is marked as a dummycr")
```

When the user types], ASKUSER just prints Nowherecr, i.e., the] is not echoed. If the user types ?, the explanation corresponding to this entry will be:

] - nowhere, item is marked as a dummy

KEYSTRING — If non-NIL, characters that are matched are echoed as though the value of KEYSTRING were used in place of the key. KEYSTRING is also used for computing the value returned. The main reason for this feature is to enable echoing in lowercase.

PROMPTON — If non-NIL, PROMPTSTRING is printed only when the key is confirmed with a member of the value of PROMPTON.

COMPLETEON — When a confirming character is typed, the N characters that are automatically supplied, as specified in case (4), are echoed only when the key is confirmed with a member of the value of PROMPTON.

The PROMPTON and COMPLETEON options enable the user to construct a KEYLST which will cause ASKUSER to emulate the action of the TENEX exec. The protocol followed by the TENEX exec is that the user can type as many characters as he likes in specifying a command. The command can be completed with a carriage return or space, in which case no further output is forthcoming, or with a $ (escape), in which case the rest of the characters in the command are echoed, followed by some prompting information. The following KEYLST would handle the TENEX COPY and CONNECT comands:

```
((COPY " (FILE LIST) "
       PROMPTON ($)
       COMPLETEON ($)
       CONFIRMFLG ($))

 (CONNECT " (TO DIRECTORY) "
```

```
PROMPTON ($)
COMPLETEON ($)
CONFIRMFLG ($)))
```

AUTOCOMPLETEFLG — If the value of the AUTOCOMPLETEFLG option is not NIL, ASKUSER will automatically supply unambiguous characters whenever it can, i.e., ASKUSER acts as though $ (escape) were typed after each character (except that it does not ring the bell if there are no unambiguous characters).

MACROCHARS — value is a list of dotted pairs of form (CHARACTER . FORM). When CHARACTER is typed, and it does not match any of the current keys, FORM is evaluated and nothing else happens, i.e. the matching process stays where it is. For example, ? could have been implemented using this option. Essentially MACROCHARS provides a read macro facility while inside of ASKUSER (since ASKUSER does READC's, read macros defined via the readtable are never invoked).

EXPLAINDELIMITER — value is what is printed to delimit explanation in response to ?. Initially a carriage return, but can be reset, e.g. to a comma, for more linear output.

Operation

All input operations are executed with the terminal table in the variable ASKUSERTTBL, in which the following is true:

- (CONTROL T) has been executed (see the Line-Buffering section of Chapter 30), so that ASKUSER can interact with the user after each character is typed

  (ECHOMODE NIL) has been executed (see the Terminal Control Functions section of Chapter 30), so that ASKUSER can decide after it reads a character whether or not the character should be echoed, and with what, e.g. unacceptable inputs are never echoed.

As each character is typed, it is matched against KEYLST, and appropriate echoing and/or prompting is performed. If the user types an unacceptable character, ASKUSER simply rings the bell and allows him to try again.

At any point, the user can type ? and receive a list of acceptable responses at that point (generated from KEYLST), or type a Control-A, Control-Q, Control-X, or delete, which causes ASKUSER to reinitialize, and start over.

Note that ?, Control-A, Control-Q, and Control-X will not work if they are acceptable inputs, i.e., they match one of the keys on KEYLST. Delete will not work if it is an interrupt character, in which case it is not seen by ASKUSER.

When an acceptable sequence is completed, ASKUSER returns the indicated value.

Completing a Key

The decision about when a key is complete is more complicated than simply whether or not all of its characters have been matched. In the compiler questions example above, all of the characters in the S key are matched as soon as the S has been typed, but until the next character is typed, ASKUSER does not know whether the S completes the S key, or is simply the first character in the ST key. Therefore, a key is considered to be complete when:

1. All of its characters have been matched and it is the only key left, i.e., there are no other keys for which this key is a substring.

2. All of its characters have been matched and a confirming character is typed.

3. All of its characters have been matched, and the value of the CONFIRMFLG option is NIL, and the value of the KEYLST option is not NIL, and the next character matches one of the keys on the value of the KEYLST option.

4. There is only one key left and a confirming character is typed. Note that if the value of CONFIRMFLG is T, the key still has to be confirmed, regardless of whether or not it is complete. For example, if the first entry in the above example were instead

   ```
   (ST "ore and redefine " CONFIRMFLG T KEYLST ("" (F . "orget exprs"))
   ```

   and the user wanted to specify the STF path, he would have to type ST, then confirm before typing F, even though the ST completed the ST key by the rule in Case 1. However, he would be prompted with ore and redefine as soon as he typed the T, and completed the ST key.

Case 2 says that confirmation can be used to complete a key in the case where it is a substring of another key, even where the value of CONFIRMFLG is NIL. In this case, the confirming character doubles as both an indicator that the key is complete, and also to confirm it, if necessary. This situation corresponds to typing Scr in the above example.

Case 3 says that if there were another entry whose key was STX in the above example, so that after the user typed ST, two keys, ST and STX, were still active, then typing F would complete the ST key, because F matches the (F . "orget exprs") entry on the value of the KEYLST option of the ST entry. In this case, ore and redefine would be printed before the F was echoed.

Finally, Case 4 says that the user can use confirmation to specify completion when only one key is left, even when all of its characters have not been matched. For example, if the first key in the above example were STORE, the user could type ST and then confirm, and ORE would be echoed, followed by whatever prompting was specified. In this case, the confirming character also confirms the key if necessary, so that no further action is required, even when the value of CONFIRMFLG is T.

Case 4 permits the user not to have to type every character in a key when the key is the only one left. Even when there are several active keys, the user can type <escape> to specify the next N>0 common characters among the currently active keys. The effect is exactly the same as though these characters had been typed. If there are no common characters in the active keys at that point, i.e. N = 0, the $ is treated as an incorrect input, and the bell is rung. For example, if KEYLST is (CLISPFLG CLISPIFYPACKFLG CLISPIFTRANFLG), and the user types C followed by $, ASKUSER will supply the L, I, S, and P. The user can then type F followed by a carriage return or space to complete and confirm CLISPFLG, as per Case 4, or type I, followed by $, and ASKUSER will supply the F, etc. Note that the characters supplied do not have to correspond to a terminal segment of any of the keys. Note also that the $ does not confirm the key, although it may complete it in the case that there is only one key active.

If the user types a confirming character when several keys are left, the next N>0 common characters are still supplied, the same as with $. However, ASKUSER assumes the intent was to complete a key, i.e., Case 4 is being invoked. Therefore, after supplying the next N characters, the bell is rung to indicate that the operation was not completed. In other words, typing a confirming character has the same effect as typing an $ in that the next N common characters are supplied. Then, if there is only one key left, the key is complete (Case 4) and confirmation is not required. If the key is not the only key left, the bell is rung.

Special Keys

& This can be used as a key to match with any single character, provided the character does not match with some other key at that level. For the purposes of echoing and returning a value, the effect is the same as though the character that were matched actually appeared as the key.

<escape> This can be used as a key to match with the result of a single call to READ. For example, if the KEYLST were:

```
((COPY " (FILE LIST) "
       PROMPTON ($)
       COMPLETEON ($)
       CONFIRMFLG ($)
       KEYLST (($ NIL RETURN ANSWER))))
```

then if the user typed COP FOOcr, (COPY FOO) would be returned as the value of ASKUSER. One advantage of using $, rather than having the calling program perform the READ, is that the call to READ from inside ASKUSER is ERRORSET protected, so that the user can back out of this path and reinitialize ASKUSER, e.g. to change from a COPY command to a CONNECT command, simply by typing Control-E.

Escape Escape This can be used as a key to match with the result of a single call to READLINE.

A list A list can be used as a key, in which case the list/form is evaluated and its value "matches" the key. This feature is provided primarily as an escape hatch for including arbitrary input operations as part of an ASKUSER sequence. For example, the effect of $$ (escape, escape) could be achieved simply by using (READLINE T) as a key.

"" The empty string can be used as a key. Since it has no characters, all of its characters are automatically matched. "" essentially functions as a place marker. For example, one of the entries on the KEYLST used by ADDTOFILES? is:

```
("" "File/list: "
EXPLAINSTRING "a file name or name of a
function list"
```

```
KEYLST ($))
```

Thus, if the user types a character that does not match any of the other keys on the KEYLST, then the character completes the "" key, by virtue of case (4), since the character will match with the $ in the inner KEYLST. ASKUSER then prints File/list: before echoing the character, then calls READ. The character will be read as part of the READ. The value returned by ASKUSER will be the value of the READ.

Note: For Escape, Escape Escape, or a list, if the last character read by the input operation is a separator, the character is treated as a confirming character for the key. However, if the last character is a break character, it will be matched against the next key.

Startup Protocol and Typeahead

Interlisp permits and encourages the user to typeahead; in actual practice, the user frequently does this. This presents a problem for ASKUSER. When ASKUSER is entered and there has been typeahead, was the input intended for ASKUSER, or was the interaction unanticipated, and the user simply typing ahead to some other program, e.g. the programmer's assistant? Even where there was no typeahead, i.e., the user starts typing after the call to ASKUSER, the question remains of whether the user had time to see the message from ASKUSER and react to it, or simply began typing ahead at an inauspicious moment. Thus, what is needed is an interlock mechanism which warns the user to stop typing, gives him a chance to respond to the warning, and then allows him to begin typing to ASKUSER.

Therefore, when ASKUSER is first entered, and the interaction is to take place with a terminal, and typeahead to ASKUSER is not permitted, the following protocol is observed:

1. If there is typeahead, ASKUSER clears and saves the input buffers and rings the bell to warn the user to stop typing. The buffers will be restored when ASKUSER completes operation and returns.

2. If MESS, the message to be printed on entry, is not NIL (the typical case), ASKUSER then prints MESS if it is a string, otherwise CAR of MESS, if MESS is a list.

3. After printing MESS or CAR of MESS, ASKUSER waits until the output has actually been printed on the terminal to make sure that the user has actually had a chance to see the output. This also give the user a chance to react. ASKUSER then checks to see if anything additional has been typed in the intervening period since it first warned the user in (1). If something has been typed, ASKUSER clears it out and again rings the bell. This latter material, i.e., that typed between the entry to ASKUSER and this point, is discarded and will not be restored since it is not certain whether the user simply reacted quickly to the first warning (bell) and this input is intended for ASKUSER, or whether the user was in the process of typing ahead when the call to ASKUSER occurred, and did not stop typing at the first warning, and therefore this input is a continuation of input intended for another program.

   Anything typed after (3) is considered to be intended for ASKUSER, i.e., once the user sees MESS or CAR of MESS, he is free to respond. For example, UNDO (see Chapter 13) calls ASKUSER when the number of undosaves are exceeded for an event with MESS = (LIST NUMBER-UNDOSAVES "undosaves, continue saving"). Thus, the user can type a response as soon as NUMBER-UNDOSAVES is typed.

4. ASKUSER then types the rest of MESS, if any.

5. Then ASKUSER goes into a wait loop until something is typed. If WAIT, the wait time, is not NIL, and nothing is typed in WAIT seconds, ASKUSER will type "..." and treat the elements of DEFAULT, the default value, as a list of characters, and begin processing them exactly as though they had been typed. If the user does type anything within WAIT seconds, he can then wait as long as he likes, i.e., once something has been typed, ASKUSER will not use the default value specified in DEFAULT.

   If the user wants to consider his response for more than WAIT seconds, and does not want ASKUSER to default, he can type a carriage return or a space, which are ignored if they are not specified as acceptable inputs by KEYLST (see below) and they are the first thing typed.

   If the calling program knows that the user is expecting an interaction with ASKUSER, e.g., another interaction preceded this one, it can specify in the call to ASKUSER that typeahead is permitted. In this case, ASKUSER simply notes whether there is any typeahead, then prints MESS and goes into a wait loop as described above.

   If there is typeahead that contains unacceptable input, ASKUSER will assume that the typeahead was not intended for ASKUSER, and will restore the typeahead when it completes operation and returns.

6. Finally, if the interaction is not with the terminal, i.e., the optional input file/string is specified, ASKUSER simply prints MESS and begins reading from the file/string.

TTYIN Display Typein Editor

TTYIN is an Interlisp function for reading input from the terminal. It features altmode completion, spelling correction, help facility, and fancy editing, and can also serve as a glorified free text input function. This document is divided into two major sections: how to use TTYIN from the user's point of view, and from the programmer's.

TTYIN exists in implementations for Interlisp-10 and Interlisp-D. The two are substantially compatible, but the capabilities of the two systems differ (Interlisp-D has a more powerful display and allows greater access to the system primitives needed to control it effectively; it also has a mouse, greatly reducing the need for keyboard-oriented editing commands). Descriptions of both are included in this document for completeness, but Interlisp-D users may find large sections irrelevant.

Entering Input With TTYIN

There are two major ways of using TTYIN: set LISPXREADFN to TTYIN, so the LISPX executive uses it to obtain input; and call TTYIN from within a program to gather text input. Mostly the same rules apply to both; places where it makes a difference are mentioned below.

The following characters may be used to edit your input, independent of what kind of terminal you are on. The more TTYIN knows about your terminal, of course, the nicer some of these will behave. Some functions are performed by one of several characters; any character that you happen to have assigned as an interrupt character will, of couse, not be read by TTYIN. There is a (somewhat inelegant) way of changing which characters perform which functions, described under TTYINREADMACROS later on.

Control-A
BACKSPACE
DELETE Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.

Control-W Deletes a "word". Generally this means back to the last space or parenthesis.

Control-Q Deletes the current line, or if the current line is blank, deletes the previous line.

Control-R Refreshes the current line. Two in a row refreshes the whole buffer (when doing multi-line input).

ESCAPE Tries to complete the current word from the spelling list provided to TTYIN, if any. In the case of ambiguity, completes as far as is uniquely determined, or rings the bell. For LISPX input, the spelling list may be USERWORDS (see discussion of TTYINCOMPLETEFLG.

Interlisp-10 only: If no spelling list was provided, but the word begins with a "<", tries directory name completion (or filename completion if there is already a matching ">" in the current word).

? If typed in the middle of a word will supply alternative completions from the SPLST argument to TTYIN (if any). ?ACTIVATEFLG (see the Assorted Flags section below) must be true to enable this feature.

Control-Y Escapes to a Lisp user exec, from which you may return by the command OK. However, when in READ mode and the buffer is non-empty, Control-Y is treated as Lisp's unquote macro instead, so you have to use meta-Control-Y (below) to invoke the user exec.

LF in Interlisp-10 Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed at TTYIN; when typed in the middle of a line fills in the remaining text from the old line; when typed following ↑Q or ↑W restores what those commands erased.

; If typed as the first character of the line means the line is a comment; it is ignored, and TTYIN loops back for more input.

Note: The exact behaviour of this character is determined by the value of TTYINCOMMENTCHAR (see the Assorted Flags section below).

Control-X Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced, beeps if not.

During most kinds of input, TTYIN is in "autofill" mode: if a space is typed near the right margin, a carriage return is simulated to start a new line. In fact, on cursor-addressable displays, lines are always broken, if possible, so that no word straddles the end of the line. The "pseudo-carriage return" ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You won't get carriage returns in your strings unless you explicitly type them.

Mouse Commands

The mouse buttons are interpreted as follows during TTYIN input:

LEFT Moves the caret to where the cursor is pointing. As you hold down LEFT, the caret moves around with the cursor; after you let up, any typein will be inserted at the new position.

MIDDLE Like LEFT, but moves only to word boundaries.

RIGHT Deletes text from the caret to the cursor, either forward or backward. While you hold down RIGHT, the text to be deleted is complemented; when you let up, the text actually goes away. If you let up outside the scope of the text, nothing is killed (this is how to "cancel" the command). This is roughly the same as CTRL-RIGHT with no initial selection (below).

If you hold down CTRL and/or SHIFT while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. You make a selection by bugging LEFT (to select a character) or MIDDLE (to select a word), and optionally extend the selection either left or right using RIGHT. While you are doing this, the caret does not move, but your selected text is highlighted in a manner indicating what is about to happen. When you have made your selection (all mouse buttons up now), lift up on CTRL and/or SHIFT and the action you have selected will occur, which is:

SHIFT	The selected text as typein at the caret. The text is highlighted with a broken underline during selection.
CTRL	Delete the selected text. The text is complemented during selection.
CTRL-SHIFT	Combines the above: delete the selected text and insert it at the caret. This is how you move text about.

You can cancel a selection in progress by pressing LEFT or MIDDLE as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing Middle-blank key (on the Xerox 1132) or the Open key (on the Xerox 1108). This is the same key that retrieves the previous buffer when issued at the end of a line.

Display Editing Commands

On terminals with a meta key: In Interlisp-10, TTYIN reads from the terminal in binary mode, allowing many more editing commands via the meta key, in the style of TVEDIT commands. Note that due to Tenex's unfortunate way of handling typeahead, it is not possible to type ahead edit commands before TTYIN has started (i.e., before its prompt appears), because the meta bit will be thrown away. Also, since Escape has numerous other meanings in Lisp and even in TTYIN (for completion), this is not used as a substitute for the meta key.

In Interlisp-D: Users will probably have little use for most of these commands, as cursor positioning can often be done more conveniently, and certainly more obviously, with the mouse. Nevertheless, some commands, such as the case changing commands, can be useful. The <bottom-blank> key can be used as an meta key if you perform (METASHIFT T) (see Chapter 30). Alternatively, you can use the variable EDITPREFIXCHAR as described in the next paragraph.

On display terminals without a meta key: If you want to type any of these commands, you need to prefix them with the "edit prefix" character. Set the variable EDITPREFIXCHAR to the character code of the desired prefix char. Type the edit prefix twice to give an "meta-escape" command. Some users of the TENEX TVEDIT program like to make escape (33Q) be the edit prefix, but this makes it somewhat awkward to ever use escape completion. EDITPREFIXCHAR is initially NIL.

On hardcopy terminals without a meta key: You probably want to ignore this section, since you won't be able to see what's going on when you issure edit commands; there is no attempt made to echo anything reasonable.

In the descriptions below, "current word" means the word the cursor is under, or if under a space, the previous word. Currently parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands. Most commands can be preceded by numbers or escape (means infinity), only the first of which requires the meta key (or the edit prefix). Some commands also accept negative arguments, but some only look at the magnitude of the arg.

Most of these commands are taken from the display editors `TVEDIT` and/or `E`, and are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands:

Meta-DELETE
Meta-BS
Meta-< Back up one (or n) characters.

Meta-SPACE
Meta-> Moves forward one (or n) characters.

Meta-^ Moves up one (or n) lines.

Meta-lf Moves down one (or n) lines.

Meta-(Moves back one (or n) words.

Meta-) Moves ahead one (or n) words.

Meta-TAB Moves to end of line; with an argument moves to nth end of line; **Meta-ESC-TAB** goes to end of buffer.

Control-Meta-L Moves to start of line (or nth previous, or start of buffer).

Meta-{
Meta-} Go to start and end of buffer, respectively.

Meta-[Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under "Flags".)

Meta-] Moves to end of current list.

Meta-Sx Skips ahead to next (or nth) occurrence of character x, or rings the bell.

Meta-Bx Backward search.

Buffer Modification Commands:

Meta-Zx Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command yet.

Meta-A
Meta-R Repeat the last S, B or Z command, regardless of any intervening input (note this differs from TEdit's A command).

Meta-K Kills the character under the cursor, or n chars starting at the cursor.

Meta-CR When the buffer is empty is the same as **LF**, i.e. restores buffer's previous contents. Otherwise is just like a **CR** (except that it also terminates an insert). Thus, **Meta-CR CR** will repeat the previous input (as will **LF CR** without the meta key).

Meta-O Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

Meta-T Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle funny cases, such as tabs.

Meta-G Grabs the contents of the previous line from the cursor position onward. Meta-nG grabs the nth previous line.

Meta-L Lowercases current word, or n words on line. **Meta-ESC-L** lowercases the rest of the line, or if given at the end of line lowercases the entire line.

Meta-U Uppercases analogously.

Meta-C Capitalize. If you give it an argument, only the first word is capitalized; the rest are just lowercased.

Control-Meta-Q Deletes the current line. **Control-Meta-ESC-Q** deletes from the current cursor position to the end of the buffer. No other arguments are handled.

Control-Meta-W Deletes the current word, or the previous word if sitting on a space.

Meta-J "Justify" this line. This will break it if it is too long, or move words up from the next line if too short. Will not join to an empty line, or one starting with a tab (both of which are interpreted as paragraph breaks). Any new line breaks it introduces are considered spaces, not carriage returns. **Meta-nJ** justifies n lines.

The linelength is defined as TTYJUSTLENGTH, ignoring any prompt characters at the margin. If TTYJUSTLENGTH is negative, it is interpreted as relative to the right margin. TTYJUSTLENGTH is initially -8 in Interlisp-D, 72 in Interlisp-10.

Meta-ESC-F "Finishes" the input, regardless of where the cursor is. Specifically, it goes to the end of the input and enters a **CR, control-Z** or "", depending on whether normal, REPEAT or READ input is happening. Note that a "" won’t necessarily end a READ, but it seems likely to in most cases where you would be inclined to use this command, and makes for more predictable behavior.

Miscellaneous Commands:

Meta-P Interlisp-D: Prettyprint buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.

Meta-N Refresh line. Same as **Control-R**. Meta-ESC-N refreshes the whole buffer; Meta-nN refreshes n lines. Cursor movement in TTYIN depends on TTYIN being the only source of output to the screen; if you do a **Control-T**, or a system message appears, or line noise occurs, you may need to refresh the line for best results. In Interlisp-10, if for some reason your terminal falls out of binary mode (e.g. can happen when returning to a Lisp running in a lower fork), **Meta-**<anything> is unreadable, so you’d have to type **Control-R** instead.

Control-Meta-Y Gets user exec. Thus, this is like regular **Control-Y**, except when doing a READ (when control-Y is a read macro and hence does not invoke this function).

Control-Meta-ESC-Y Gets a user exec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for the Lisp executive, you can do **Control-Meta-L-ESC-Control-Y** and give it to Lisp.

Meta-← Adds the current word to the spelling list USERWORDS. With zero arg, removes word. See TTYINCOMPLETEFLG (see the Assorted Flags section below).

Note to Datamedia, Heath users: In addition to simple cursor movement commands and insert/delete, TTYIN uses the display's cursor-addressing capability to optimize cursor movements longer than a few characters, e.g. **Meta-TAB** to go to the end of the line. In order to be able to address the cursor, TTYIN has to know where it is to begin with. Lisp keeps track of the current print position within the line, but does not keep track of the line on the screen (in fact, it knows precious little about displays, much like Tenex). Thus, TTYIN establishes where it is by forcing the cursor to appear on the last line of the screen. Ordinarily this is the case anyway (except possibly on startup), but if the cursor happens to be only halfway down the screen at the time, there is a possibly unsettling leap of the cursor when TTYIN starts.

Using TTYIN for Lisp Input

When TTYIN is loaded, or a sysout containing TTYIN is started up, the function SETREADFN is called. If the terminal is a display, it sets LISPXREADFN (see Chapter 13) to be TTYINREAD. If the terminal is not a display terminal, SETREADFN will set the variable to READ. (SETREADFN 'READ) will also set it to READ.

There are two principal differences between TTYINREAD and READ: (1) parenthesis balancing. The input does not activate on an exactly balancing right paren/bracket unless the input started with a paren/bracket, e.g., USE (FOO) FOR (FIE) will all be on one line, terminated by **CR**; and (2) read macros.

In Interlisp-10, TTYIN does not use a read table (TTYIN behaves as though using the default initial Lisp terminal input readtable), so read macros and redefinition of syntax characters are not supported; however, " ' " (QUOTE) and "Control-Y" (EVAL) are built in, and a simple implementation of ? and ?= is supplied. Also, the TTYINREADMACROS facility described below can supply some of the functionality of immediate read macros in the editor.

In Interlisp-D, read macros are (mostly) supported. Immediate read macros take effect only if typed at the end of the input (it's not clear what their semantics should be elsewhere).

Useful Macros

There are two useful edit macros that allow you to use TTYIN as a character editor: (1) ED loads the current expression into the ttyin buffer to be edited (this is good for editing comments and strings). Input is terminated in the usual way (by typing a balancing right programmer's assistant command FIX will load the buffer with the event's input, rather than calling the editor. If you really wanted the

Interlisp editor for your fix, you can say FIX EVENT - TTY: once you got TTYIN's version to force you into the editor.

Programming With TTYIN

(**TTYIN** PROMPT SPLST HELP OPTIONS ECHOTOFILE TABS UNREADBUF RDTBL) [Function]

TTYIN prints PROMPT, then waits for input. The value returned in the normal case is a list of all atoms on the line, with comma and parens returned as individual atoms; OPTIONS may be used to get a different kind of value back.

PROMPT is an atom or string (anything else is converted to a string). If NIL, the value of DEFAULTPROMPT, initially "** ", will be used. If PROMPT is T, no prompt will be given. PROMPT may also be a dotted pair (PROMPT1 . PROMPT2), giving the prompt for the first and subsequent (or overflow) lines, each prompt being a string/atom or NIL to denote absence of prompt. The default prompt for overflow lines is "...". Note that rebinding DEFAULTPROMPT gives a convenient way to affect all the "ordinary" prompts in some program module.

SPLST is a spelling list, i.e., a list of atoms or dotted pairs (SYNONYM . ROOT). If supplied, it is used to check and correct user responses, and to provide completion if the user types escape. If SPLST is one of the Lisp system spelling lists (e.g., USERWORDS or SPELLINGS3), words that are escape-completed get moved to the front, just as if a FIXSPELL had found them. Autocompletion is also performed when user types a break character (cr, space, paren, etc), unless one of the "nofixspell" options below is selected; i.e., if the word just typed would uniquely complete by escape, TTYIN behaves as though escape had been typed.

HELP, if non-NIL, determines what happens when the user types ? or HELP. If HELP = T, program prints back SPLST in suitable form. If HELP is any other litatom, or a string containing no spaces, it performs (DISPLAYHELP HELP). Anything else is printed as is. If HELP is NIL, ? and HELP are treated as any other atoms the user types. [DISPLAYHELP is a user-supplied function, initially a noop; systems with a suitable HASH package, for example, have defined it to display a piece of text from a hashfile associated with the key HELP.]

OPTIONS is an atom or list of atoms chosen from among the following:

NOFIXSPELL — Uses SPLST for HELP and Escape completion, but does not attempt any FIXSPELLing. Mainly useful if SPLST is incomplete and the caller wants to handle corrections in a more flexible way than a straight FIXSPELL.

MUSTAPPROVE — Does spelling correction, but requires confirmation.

CRCOMPLETE — Requires confirmation on spelling correction, but also does autocompletion on <cr> (i.e. if what user has typed so far uniquely identifies a member of SPLST, completes it). This allows you to have the benefits of autocompletion and still allow new words to be typed.

DIRECTORY (only if SPLST = NIL) Interprets Escape to mean directory name completion [Interlisp-10 only].

USER Like DIRECTORY, but does username completion. This is identical to DIRECTORY under Tenex [Interlisp-10 only].

FILE (only if SPLST = NIL) Interprets Escape to mean filename completion [Sumex and Tops20 only].

FIX If response is not on, or does not correct to, SPLST, interacts with user until an acceptable response is entered. A blank line (returning NIL) is always accepted. Note that if you are willing to accept responses that are not on SPLST, you probably should specify one of the options NOXFISPELL, MUSTAPPROVE or CRCOMPLETE, lest the user's new response get FIXSPELLed away without their approval.

STRING Line is read as a string, rather than list of atoms. Good for free text.

NORAISE Does not convert lower case letters to upper case.

NOVALUE For use principally with the ECHOTOFILE arg (below). Does not compute a value, but returns T if user typed anything, NIL if just a blank line.

REPEAT For multi-line input. Repeatedly prompts until user types Control-Z (as in Tenex sndmsg). Returns one long list; with STRING option returns a single string of everything typed, with carriage returns (EOL) included in the string.

TEXT Implies REPEAT, NORAISE, and NOVALUE. Additionally, input may be terminated with Control-V, in which case the global flag CTRLVFLG will be set true (it is set to NIL on any other termination). This flag may be utilized in any way the caller desires.

COMMAND Only the first word on the line is treated as belonging to SPLST, the remainder of the line being arbitrary text; i.e., "command format". If other options are supplied, COMMAND still applies to the first word typed. Basically, it always returns (CMD . REST-OF-INPUT), where REST-OF-INPUT is whatever the other options dictate for the remainder. E.g. COMMAND NOVALUE returns (CMD) or (CMD . T), depending on whether there was further input; COMMAND STRING returns (CMD . "REST-OF-INPUT"). When used with REPEAT, COMMAND is only in effect for the first line typed; furthermore, if the first line consists solely of a command, the REPEAT is ignored, i.e., the entire input is taken to be just the command.

READ Parens, brackets, and quotes are treated a la READ, rather than being returned as individual atoms. Control characters may be input via the Control-Vx notation. Input is terminated roughly along the lines of READ conventions: a balancing or over-balancing right paren/bracket will activate the input, or <cr> when no parenthesis remains unbalanced. READ overrides all other options (except NORAISE).

LISPXREAD Like READ, but implies that TTYIN should behave even more like READ, i.e., do NORAISE, not be errorset-protected, etc.

NOPROMPT Interlisp-D only: The prompt argument is treated as usual, except that TTYIN assumes that the prompt for the first line has already been printed by the caller; the prompt for the first line is thus used only when redisplaying the line.

ECHOTOFILE if specified, user's input is copied to this file, i.e., TTYIN can be used as a simple text-to-file routine if NOVALUE is used. If ECHOTOFILE is a list, copies to all files in the list. PROMPT is not included on the file.

TABS is a special addition for tabular input. It is a list of tabstops (numbers). When user types a tab, TTYIN automatically spaces over to the next tabstop (thus the first tabstop is actually the second "column" of input). Also treats specially the characters * and "; they echo normally, and then automatically tab over.

UNREADBUF allows the caller to "preload" the TTYIN buffer with a line of input. UNREADBUF is a list, the elements of which are unread into the buffer (i.e., "the outer parentheses are stripped off") to be edited further as desired; a simple carriage return (or Control-Z for REPEAT input) will thus cause the buffer's contents to be returned unchanged. If doing READ input, the "PRIN2 names" of the input list are used, i.e., quotes and %'s will appear as needed; otherwise the buffer will look as though UNREADBUF had been PRIN1'ed. UNREADBUF is treated somewhat like READBUF, so that if it contains a pseudo-carriage return (the value of HISTSTR0), the input line terminates there.

Input can also be unread from a file, using the HISTSTR1 format: UNREADBUF = (<value of HISTSTR1> (FILE START . END)), where START and END are file byte pointers. This makes TTYIN a miniature text file editor.

RDTBL [Interlisp-D only] is the read table to use for READing the input when one of the READ options is given. A lot of character interpretations are hardwired into TTYIN, so currently the only effect this has is in the actual READ, and in deciding whether a character typed at the end of the input is an immediate read macro, for purposes of termination.

If the global variable TYPEAHEADFLG is T, or option LISPXREAD is given, TTYIN permits type-ahead; otherwise it clears the buffer before prompting the user.

Using TTYIN as a General Editor

The following may be useful as a way of outsiders to call TTYIN as an editor. These functions are currently only in Interlisp-D.

(**TTYINEDIT** *EXPRS WINDOW PRINTFN PROMPT*) [Function]

This is the body of the edit macro EE. Switches the tty to *WINDOW*, clears it, prettyprints *EXPRS*, a list of expressions, into it, and leaves you in TTYIN to edit it as Lisp input. Returns a new list of expressions.

If *PRINTFN* is non-NIL, it is a function of two arguments, EXPRS and FILE, which is called instead of PRETTYPRINT to print the expressions to the window (actually to a scratch file). Note that EXPRS is a list, so normally the outer parentheses should not be printed. *PRINTFN* = T is shorthand for "unpretty"; use PRIN2 instead of PRETTYPRINT.

PROMPT determines what prompt is printed, if any. If T, no prompt is printed. If NIL, it defaults to the value of TTYINEDITPROMPT.

TTYINAUTOCLOSEFLG [Variable]

If TTYINAUTOCLOSEFLG is true, TTYINEDIT closes the window on exit.

TTYINEDITWINDOW [Variable]

If the WINDOW arg to TTYINEDIT is NIL, it uses the value of TTYINEDITWINDOW, creating it if it does not yet exist.

TTYINPRINTFN [Variable]

The default value for PRINTFN in EE's call to TTYINEDIT.

(**SET.TTYINEDIT.WINDOW** *WINDOW*) [Function]

Called under a RESETLST. Switches the tty to *WINDOW* (defaulted as in TTYINEDIT) and clears it. The window's position is left so that TTYIN will be happy with it if you now call TTYIN yourself. Specifically, this means positioning an integral number of lines from the bottom of the window, the way the top-level tty window normally is.

(**TTYIN.SCRATCHFILE**) [Function]

Returns, possibly creating, the scratchfile that TTYIN uses for prettyprinting its input. The file pointer is set to zero. Since TTYIN does use this file, beware of multiple simultaneous use of the file.

?= Handler

In Interlisp, the ?= read macro displays the arguments to the function currently "in progress" in the typein. Since TTYIN wants you to be able to continue editing the buffer after a ?=, it processes this macro specially on its own, printing the arguments below your typein and then putting the cursor back where it was when ?= was typed. For users who want special treatment of ?=, the following hook exists:

TTYIN?=FN [Variable]

The value of this variable, if non-NIL, is a user function of one argument that is called when ?= is typed. The argument is the function that ?= thinks it is inside of. The user function should return one of the following:

NIL Normal ?= processing is performed.

T Nothing is done. Presumably the user function has done something privately, perhaps diddled some other window, or called TTYIN.PRINTARGS (below).

a list (ARGS . STUFF) Treats STUFF as the argument list of the function in question, and performs the normal ?= processing using it.

anything else The value is printed in lieu of what ?= normally prints.

At the time that ?= is typed, nothing has been "read" yet, so you don't have the normal context you might expect inside a conventional readmacro. If the user function wants to examine the typed-in arguments being passed to the fn, however, it can call the function TTYIN.READ?=ARGS:

(**TTYIN.READ?=ARGS**) [Function]

When called inside TTYIN?=FN user function, returns everything between the function and the typing of ?= as a list (like an arglist). Returns NIL if ?= was typed immediately after the function name.

(**TTYIN.PRINTARGS** *FN ARGS ACTUALS ARGTYPE*) [Function]

Does the function/argument printing for ?=. *ARGS* is an argument list, *ACTUALS* is a list of actual parameters (from the typein) to match up with args. *ARGTYPE* is a value of the function *ARGTYPE*; it defaults to (*ARGTYPE FN*).

Read Macros

When doing READ input in Interlisp-10, no Lisp-style read macros are available (but the ' and control-Y macros are built in). Principally because of the usefulness of the editor read macros (set by SETTERMCHARS), and the desire for a way of changing the meanings of the display editing commands, the following exists as a hack:

TTYINREADMACROS [Variable]

Value is a set of shorthand inputs useable during READ input. It is an alist of entries (CHARCODE . SYNONYM). If the user types the indicated character (the meta bit is denoted by the 200Q bit in the char code), TTYIN behaves as though the synonym character had been typed.

Special cases: 0 - the character is ignored; 200Q - pure meta bit; means to read another char and turn on its meta bit; 400Q - macro quote: read another char and use its original meaning. For example, if you have macros ((33Q . 200Q) (30Q . 33Q)), then Escape (33Q) will behave as an edit prefix, and control-X (30Q) will behave like Escape.

Note: currently, synonyms for meta commands are not well-supported, working only when the command is typed with no argument.

Slightly more powerful macros also can be supplied; they are recognized when a character is typed on an empty line, i.e., as the first thing after the prompt. In this case, the TTYINREADMACROS entry is of the form (CHARCODE T . RESPONSE) or (CHARCODE CONDITION . RESPONSE), where CONDITION is a list that evaluates true. If RESPONSE is a list, it is EVALed; otherwise it is left unevaluated. The result of this evaluation (or RESPONSE itself) is treated as follows:

NIL — The macro is ignored and the character reads normally, i.e., as though TTYINREADMACROS had never existed.

An integer — A character code, treated as above. Special case: -1 is treated like 0, but says that the display may have been altered in the evaluation of the macro, so TTYIN should reset itself appropriately.

Anything else — This TTYIN input is terminated (with a crlf) and returns the value of "response" (turned into a list if necessary). This is the principal use of this facility. The macro character thus stands for the (possibly computed) reponse, terminated if necessary with a crlf. The original character is not echoed.

Interrupt characters, of course, cannot be read macros, as TTYIN never sees them, but any other characters, even non-control chars, are allowed. The ability to return NIL allows you to have conditional macros that only apply in specified situations (e.g., the macro might check the prompt (LISPXID) or other contextual variables). To use this specifically to do immediate editor read macros, do the following for each edit command and character you want to invoke it with:

```
(ADDTOVAR TTYINREADMACROS (CHARCODE 'CHARMACRO?
EDITCOM)))
```

For example, (ADDTOVAR TTYINREADMACROS (12Q CHARMACRO? !NX)) will make linefeed do the !NX command. Note that this will only activate linefeed at the beginning of a line, not anywhere in the line. There will probably be a user function to do this in the next release.

Note that putting (12Q T . !NX) on TTYINREADMACROS would also have the effect of returning !NX from the READ call so that the editor would do an !NX. However, TTYIN would also return !NX outside the editor (probably resulting in a u.b.a. error, or convincing DWIM to enter the editor), and also the clearing of the output buffer (performed by CHARMACRO?) would not happen.

Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. In Interlisp-D, the flags are all initially set to T.

TYPEAHEADFLG [Variable]

If true, TTYIN always permits typeahead; otherwise it clears the buffer for any but LISPXREAD input.

?**ACTIVATEFLG** [Variable]

If true, enables the feature whereby ? lists alternative completions from the current spelling list.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis/bracket, TTYIN will briefly move the cursor to the matching parenthesis/bracket, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you'll never notice it). This feature was inspired by a similar EMACS feature, and turned out to be pretty easy to implement.

TTYINBSFLG [Variable]

Causes TTYIN to always physically backspace, even if you're running on a non-display (not a DM or Heath), rather than print \deletedtext\ (this assumes your hardcopy terminal or glass tty is capable of backspacing). If TTYINBSFLG is LF, then in addition to backspacing, TTYIN x's out the deleted characters as it backs up, and when you stop deleting, it outputs a linefeed to drop to a new, clean line before resuming. To save paper, this linefeed operation is not done when only a single character is deleted, on the grounds that you can probably figure out what you typed anyway.

TTYINRESPONSES [Variable]

An association list of special responses that will be handled by routines designated by the programmer. See "Special Responses", below.

TTYINERRORSETFLG [Variable]

[Interlisp-D only] If true, non-LISPXREAD inputs are errorset-protected (Control-E traps back to the prompt), otherwise errors propagate upwards. Initially NIL.

TTYINCOMMENTCHAR [Variable]

This variable affects the treatment of lines beginning with the comment character (usually ";"). If TTYINCOMMENTCHAR is a character code, and the first character on a line of typein is equal to TTYINCOMMENTCHAR, then the line is erased from the screen and no input function will see it. If TTYINCOMMENTCHAR is NIL, this feature is disabled. TTYINCOMMENTCHAR is initially NIL.

TTYINCOMPLETEFLG [Variable]

If true, enables Escape completion from USERWORDS during READ inputs. Details below.

USERWORDS (see Chapter 20) contains words you mentioned recently: functions you have defined or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently edited a function, chances are good you may want to edit it again (typing "EF xx$") or type a call to it. If there is no completion for the current word from USERWORDS, the escape echoes as "$", i.e. nothing special happens; if there is more than one possible completion, you get beeped. If typed when not inside a word, Escape completes to the value of LASTWORD, i.e., the last thing you typed that the p.a. "noticed" (setting TTYINCOMPLETEFLG to 0 disables this latter feature), except that Escape at the beginning of the line is left alone (it is a p.a. command).

If you really wanted to enter an escape, you can, of course, just quote it with a control-V, like you can other control chars.

You may explicitly add words to USERWORDS yourself that wouldn't get there otherwise. To make this convenient online the edit command [←] means "add the current atom to USERWORDS" (you might think of the command as "pointing out this atom"). For example, you might be entering a function definition and want to "point to" one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from USERWORDS.

Note that this feature loses some of its value if the spelling list is too long, for then the completion takes too long computationally and, more important, there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list USERWORDS keeps the "temporary" section (which is where everything goes initially unless you say otherwise) limited to #USERWORDS atoms, initially 100. Words fall off the end if they haven't been used (they are "used" if FIXSPELL corrects to one, or you use <escape> to complete one).

Special Responses

There is a facility for handling "special responses" during any non-READ TTYIN input. This action is independent of the particular call to TTYIN, and exists to allow you to effectively "advise" TTYIN to intercept certain commands. After the command is processed, control returns to the original TTYIN call. The facility is implemented via the list TTYINRESPONSES.

TTYINRESPONSES [Variable]

TTYINRESPONSES is a list of elements, each of the form:

```
(COMMANDS RESPONSE-FORM OPTION)
```

COMMANDS is a single atom or list of commands to be recognized; RESPONSE-FORM is EVALed (if a list), or APPLYed (if an atom) to the command and the rest of the line. Within this form one can reference the free variables COMMAND (the command the user typed) and LINE (the rest of the line). If OPTION is the atom LINE, this means to pass the rest of line as a list; if it is STRING, this means to pass it as a string; otherwise, the command is only valid if there is nothing else on the line. If RESPONSE-FORM returns the atom IGNORE, it is not treated as a special response (i.e. the input is returned normally as the result of TTYIN).

Suggested use: global commands or options can be added to the toplevel value of TTYINRESPONSES. For more specialized commands, rebind TTYINRESPONSES to (APPEND NEWENTRIES TTYINRESPONSES) inside any module where you want to do this sort of special processing.

Special responses are not checked for during READ-style input.

Display Types

[This is not relevant in Interlisp-D]

TTYIN determines the type of display by calling DISPLAYTERMP, which is initially defined to test the value of the GTTYP jsys. It returns either NIL (for printing terminals) or a small number giving TTYIN's internal code for the terminal type. The types TTYIN currently knows about:

0 = glass tty (capable of deleting chars by backspacing, but little else)

1 = Datamedia

2 = Heath

Only the Datamedia has full editing power. DISPLAYTERMP has built into it the correct terminal types for Sumex and Stanford campus 20's: Datamedia = 11 on tenex, 5 on tops20; Heath = 18 on Tenex, 25 on tops20. You can override those values by setting the variable DISPLAYTYPES to be an association list associating the GTTYP value with one of these internal codes. For example, Sumex displays correspond to DISPLAYTYPES = ((11 . 1) (18 . 2)) [although this is actually compiled into DISPLAYTERMP for speed]. Any display terminal other than Datamedia and Heath can probably safely be assigned to "0" for glass tty.

To add new terminal types, you have to choose a number for it, add new code to TTYIN for it and recompile. The TTYIN code specifies what the capabilities of the terminal are, and how to do the primitive operations: up, down, left, right, address cursor, erase screen, erase to end of line, insert character, etc.

For terminals lacking a meta key (currently only Datamedias have it), set the variable EDITPREFIXCHAR to the ascii code of an edit "prefix" (i.e., anything typed preceded by the prefix is considered to have the meta bit on). If your EDITPREFIXCHAR is 33Q (Escape), you can type a real Escape by typing 3 of them (2 won't do, since that means "Meta-Escape", a legitimate argument to another command). You could also define an Escape synonym with TTYINREADMACROS if you wanted (but currently it doesn't work in filename completion). Setting EDITPREFIXCHAR for a terminal that is not equipped to handle the full range of editing functions (only the Heath and Datamedia are currently so equipped) is not guaranteed to work, i.e. the display will not always be up to date; but if you can keep track of what you're doing, together with an occasional control-R to help out, go right ahead.

Prettyprint

The standard way of printing out function definitions (on the terminal or into files) is to use PRETTYPRINT.

(**PRETTYPRINT** *FNS PRETTYDEFLG* —) [Function]

FNS is a list of functions. If *FNS* is atomic, its value is used). The definitions of the functions are printed in a pretty format on the primary output file using the primary readtable. For example, if FACTORIAL were defined by typing

```
(DEFINEQ (FACTORIAL [LAMBDA (N) (COND ((ZEROP N) 1)
(T (ITIMES N (FACTORIAL (SUB1 N]
```

```
(PRETTYPRINT '(FACTORIAL))would print out
(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITIMES N (FACTORIAL (SUB1 N])
```

PRETTYDEFLG is T when called from PRETTYDEF (and hence MAKEFILE). Among other actions taken when this argument is true, PRETTYPRINT indicates its progress in writing the current output file: whenever it starts a new function, it prints on the terminal the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

PRETTYPRINT operates correctly on functions that are BROKEN, BROKEN-IN, ADVISED, or have been compiled with their definitions saved on their property lists: it prints the original, pristine definition, but does not change the current state of the function. If a function is not defined but is known to be on one of the files noticed by the file package, PRETTYPRINT loads in the definition (using LOADFNS) and prints it (except when called from PRETTYDEF). If PRETTYPRINT is given an atom which is not the name of a function, but has a value, it prettyprints the value. Otherwise, PRETTYPRINT attempts spelling correction. If all fails, PRETTYPRINT returns (FN NOT PRINTABLE). Note that PRETTYPRINT will return (FN NOT PRINTABLE) if FN does not have an accessable expr definition, or if it doesn't have any definition at all.

(**PP** *FN_1 ... FN_N*) [NLambda NoSpread Function]

For prettyprinting functions to the terminal. PP calls PRETTYPRINT with the primary output file set to T and the primary read table set to T. The primary output file and primary readtable are restored after printing.

(PP FOO) is equivalent to (PRETTYPRINT '(FOO)); (PP FOO FIE) is equivalent to (PRETTYPRINT '(FOO FIE)).

As described above, when PRETTYPRINT, and hence PP, is called with the name of a function that is not defined, but whose definition is on a file known to the file package, the definition is automatically read in and then prettyprinted. However, if the user does not intend on editing or running the definition, but simply wants to see the definition, the function PF described below can be used to simply copy the corresponding characters from the file to the terminal. This results in a savings in both space and time, since it is not necessary to allocate storage to actually read in the definition, and it is not necessary to re-prettyprint it (since the function is already in prettyprint format on the file).

(**PF** *FN FROMFILES TOFILE*) [NLambda NoSpread Function]

Copies the definition of *FN* found on each of the files in *FROMFILES* to *TOFILE*. If *TOFILE* = NIL, defaults to T. If *FROMFILES* = NIL, defaults to (WHEREIS FN NIL T) (see Chapter 17). The typical usage of PF is simply to type "PF FN".

PF prints a message if it can't find a file on *FROMFILES*, or it can't find the function FN on a file.

When printing to the terminal, PF performs several transformations on the characters in the file that comprise the definition for FN:

1. Font information is stripped out (except in Interlisp-D, whose display supports multiple fonts)
2. Occurrences of the CHANGECHAR (see the Special Prettyprint Controls section below) are not printed
3. Since functions typically tend to be printed to a file with a larger linelength than when printing to a terminal, the number of leading spaces on each line is cut in half (unless PFDEFAULT is T; initially NIL)
4. Comments are elided, if **COMMENT**FLG is non-NIL (see the Comment Feature section below).

(**SEE** *FROMFILE TOFILE*) [NLambda NoSpread Function]

Copies all of the text from *FROMFILE* to *TOFILE* (defaults to T), processing all text as PF does. Used to display the contents of files on the terminal.

(**PP*** *X*) [NLambda NoSpread Function]

(**PF*** *FN FROMFILES TOFILE*) [NLambda NoSpread Function]

(**SEE*** *FROMFILE TOFILE*) [NLambda NoSpread Function]

These functions operate exactly like PP, PF, and SEE, except that they bind **COMMENT**FLG to NIL, so comments are printed in full.

While the function PRETTYPRINT prints entire function definitions, the function PRINTDEF can be used to print parts of functions, or arbitrary Interlisp structures:

(**PRINTDEF** *EXPR LEFT DEF TAILFLG FNSLST FILE*) [Function]

Prints the expression *EXPR* in a pretty format on *FILE* using the primary readtable. *LEFT* is the left hand margin (LINELENGTH determines the right hand margin). PRINTDEF initially performs (TAB LEFT T), which means to space to position *LEFT*, unless already beyond this position, in which case it does nothing.

DEF = T means *EXPR* is a function definition, or a piece of one. If *DEF* = NIL, no special action is taken for LAMBDA's, PROG's, COND's, comments, CLISP, etc. DEF is NIL when PRETTYDEF calls PRETTYPRINT to print variables and property lists, and when PRINTDEF is called from the editor via the command PPV.

TAILFLG = T means *EXPR* is interpreted as a tail of a list, to be printed without parentheses.

FNSLST is for use for printing with multiple fonts (see Chapter 27). PRINTDEF prints occurrences of any function in the list FNSLST in a different font, for emphasis. MAKEFILE passes as FNSLST the list of all functions on the file being made.

Comment Feature

A facility for annotating Interlisp functions is provided in PRETTYPRINT. Any expression beginning with the atom * is interpreted as a comment and printed in the right margin. Example:

```
(FACTORIAL
   [LAMBDA (N)                   (* COMPUTES N!)
      (COND
        ((ZEROP N)               (* 0! = 1)
             1)
        (T           (* RECURSIVE DEFINITION:
                      N! = N*N-1!)
          (ITIMES N (FACTORIAL (SUB1 N])
```

These comments actually form a part of the function definition. Accordingly, * is defined as an nlambda nospread function that returns its argument, similar to QUOTE. When running an interpreted function, * is entered the same as any other Interlisp function. Therefore, comments should only be placed where they will not harm the computation, i.e., where a quoted expression could be placed. For example, writing

```
(ITIMES N (FACTORIAL (SUB1 N)) (* RECURSIVE
DEFINITION))
```

in the above function would cause an error when ITIMES attempted to multiply N, N-1!, and RECURSIVE.

For compilation purposes, * is defined as a macro which compiles into no instructions (unless the comment has been placed where it has been used for value, in which case the compiler prints an appropriate error message and compiles * as QUOTE). Thus, the compiled form of a function with comments does not use the extra atom and list structure storage required by the comments in the source (interpreted) code. This is the way the comment feature is intended to be used.

A comment of the form (* E X) causes X to be evaluated at prettyprint time, as well as printed as a comment in the usual way. For example, (* E (RADIX 8)) as a comment in a function containing octal numbers can be used to change the radix to produce more readable printout.

The comment character * is stored in the variable COMMENTFLG. The user can set it to some other value, e.g. ";", and use this to indicate comments.

COMMENTFLG [Variable]

If CAR of an expression is EQ to COMMENTFLG, the expression is treated as a comment by PRETTYPRINT. COMMENTFLG is initialized to *. Note that whatever atom is chosen for COMMENTFLG should also have an appropriate function definition and compiler macro, for example, by copying those of *.

Comments are designed mainly for documenting listings. Therefore, when prettyprinting to the terminal, comments are suppressed and printed as the string **COMMENT**. The value of **COMMENT**FLG determines the action.

****COMMENT**FLG** [Variable]

If **COMMENT**FLG is NIL, comments are printed. Otherwise, the value of **COMMENT**FLG is printed. Initially " **COMMENT** ".

(**COMMENT1** *L* —) [Function]

Prints the comment *L*. COMMENT1 is a separate function to permit the user to write prettyprint macros that use the regular comment printer. For example, to cause comments to be printed at a larger than normal linelength, one could put an entry for * on PRETTYPRINTMACROS:

```
(* LAMBDA (X) (RESETFORM (LINELENGTH 100)
(COMMENT1 X)))
```

This macro resets the line length, prints the comment, and then restores the line length.

COMMENT1 expects to be called from within the environment established by PRINTDEF, so ordinarily the user should call it only from within prettyprint macros.

Comment Pointers

For a well-commented collection of programs, the list structure, atom, and print name storage required to represent the comments in core can be significant. If the comments already appear on a file and are not needed for editing, a significant savings in storage can be achieved by simply leaving the text of the comment on the file when the file is loaded, and instead retaining in core only a pointer to the comment. When this feature is enabled, * is defined as a read macro (see Chapter 25) in

FILERDTBL which, instead of reading in the entire text of the comment, constructs an expression containing

- The name of the file in which the text of the comment is contained
- The address of the first character of the comment
- The number of characters in the comment
- A flag indicating whether the comment appeared at the right hand margin or centered on the page

For output purposes, * is defined on PRETTYPRINTMACROS (see the Prettyprint Control Functions section below) so that it prints the comments represented by such pointers by simply copying the corresponding characters from one file to another, or to the terminal. Normal comments are processed the same as before, and can be intermixed freely with comment pointers.

The comment pointer feature is controlled by the function NORMALCOMMENTS.

(**NORMALCOMMENTS** *FLG*) [Function]

If *FLG* is NIL, the comment pointer feature is enabled. If *FLG* is T, the comment pointer feature is disabled (the default).

NORMALCOMMENTS can be changed as often as desired. Thus, some files can be loaded normally, and others with their comments converted to comment pointers.

For convenience of editing selected comments, an edit macro, GET*, is included, which loads in the text of the corresponding comment. The editor's PP* command, in contrast, prints the comment without reading it by simply copying the corresponding characters to the terminal. GET* is defined in terms of GETCOMMENT:

(**GETCOMMENT** *X* *DESTFL* —) [Function]

If *X* is a comment pointer, replaces *X* with the actual text of the comment, which it reads from its file. Returns *X* in all cases. If *DESTFL* is non-NIL, it is the name of an open file, to which GETCOMMENT copies the comment; in this case, *X* remains a comment pointer, but it has been changed to point to the new file (unless NORMALCOMMENTS has been set to DONTUPDATE).

(**PRINTCOMMENT** *X*) [Function]

Defined as the prettyprint macro for *: copies the comment to the primary output file by using GETCOMMENT.

(**READCOMMENT** *FL* *RDTBL* *LST*) [Function]

Defined as the read macro for * in FILERDTBL: if NORMALCOMMENTSFLG is NIL, it constructs a comment pointer, unless it believes the expression beginning with * is not actually a comment, e.g., if the next atom is " . " or E.

Note that a certain amount of care is required in using the comment pointer feature. Since the text of the comment resides on the file pointed to by the comment pointer, that file must remain in existence as long as the comment is needed. GETCOMMENT helps out by changing the comment pointer to always point at the most recent file that the comment lives on. However, if the user has been performing repeated MAKEFILE's (see Chapter 17) in which differing functions have changed at each invocation of MAKEFILE, it is possible for the comment pointers in memory to be pointing at several versions of the same file, since a comment pointer is only updated when the function it lives in is prettyprinted, not when the function has been copied verbatim to the new file. This can be a problem for file systems that have a built-in limit on the number of versions of a given file that will be made before old versions are expunged. In such a case, the user should set the version retention count of any directories involved to be infinite. GETCOMMENT prints an error message if the file that the comment pointer points at has disappeared.

Similarly, one should be cognizant of comment pointers in sysouts, and be sure to retain any files thus pointed to.

When using comment pointers, the user should also not set PRETTYFLG to NIL or call MAKEFILE with option FAST, since this will prevent functions from being prettyprinted, and hence not get the text of the comment copied into the new file.

If the user changes the value of COMMENTFLG but still wishes to use the comment pointer feature, the new COMMENTFLG should be given the same read-macro definition in FILERDTBL as * has, and the same entry be put on PRETTYPRINTMACROS. For example, if COMMENTFLG is reset to be ";", then (SETSYNTAX '; '* FILERDTBL) should be performed, and (; . PRINTCOMMENT) added to PRETTYPRINTMACROS.

Converting Comments to Lower Case

This section is for users using terminals without lower case, who nevertheless would like their comments to be converted to lower case for more readable listings. If the second atom in a comment is %%, the text of the comment is converted to lower case so that it looks like English instead of Lisp. Note that comments are converted only when they are actually written to a file by PRETTYPRINT.

The algorithm for conversion to lower case is the following: If the first character in an atom is ^, do not change the atom (but remove the ^). If the first character is %, convert the atom to lower case. Note that the user must type %% as % is the escape character. If the atom (minus any trailing punctuation marks) is an Interlisp word (i.e., is a bound or free variable for the function containing the comment, or has a top level value, or is a defined function, or has a non-NIL property list), do not change it. Otherwise, convert the atom to lower case. Conversion only affects the upper case alphabet, i.e., atoms already converted to lower case are not changed if the comment is converted again. When converting, the first character in the comment and the first character following each period are left capitalized. After conversion, the comment is physically modified to be the lower case text minus the %% flag, so that conversion is thus only performed once (unless the user edits the comment inserting additional upper case text and another %% flag).

LCASELST [Variable]

Words on LCASELST will always be converted to lower case. LCASELST is initialized to contain words which are Interlisp functions but also appear frequently in comments as English words (AND, EVERY, GET, GO, LAST, LENGTH, LIST, etc.). Therefore, if one wished to type a comment including the lisp fuction GO, it would be necessary to type ↑GO in order that it might be left in upper case.

UCASELST [Variable]

Words on UCASELST (that do not appear on LCASELST) will be left in upper case. UCASELST is initialized to NIL.

ABBREVLST [Variable]

ABBREVLST is used to distinguish between abbreviations and words that end in periods. Normally, words that end in periods and occur more than halfway to the right margin cause carriage-returns. Furthermore, during conversion to lowercase, words ending in periods, except for those on ABBREVLST, cause the first character in the next word to be capitalized. ABBREVLST is initialized to the upper and lower case forms of ETC., I.E., and E.G..

Special Prettyprint Controls

PRETTYTABFLG [Variable]

In order to save space on files, tabs are used instead of spaces for the inital spaces on each line, assuming that each tab corresponds to 8 spaces. This results in a reduction of file size by about 30%. Tabs are not used if PRETTYTABFLG is set to NIL (initially T).

#RPARS [Variable]

Controls the number of right parentheses necessary for square bracketing to occur. If #RPARS = NIL, no brackets are used. #RPARS is initialized to 4.

FIRSTCOL [Variable]

The starting column for comments. Comments run between FIRSTCOL and the line length set by LINELENGTH (see Chapter 25). If a word in a comment ends with a "." and is not on the list ABBREVLST, and the position is greater than halfway between FIRSTCOL and LINELENGTH, the next word in the comment begins on a new line. Also, if a list is encountered in a comment, and the position is greater than halfway, the list begins on a new line.

PRETTYLCOM [Variable]

If a comment has more than PRETTYLCOM elements (using COUNT), it is printed starting at column 10, instead of FIRSTCOL. Comments are also printed starting at column 10 if their second element is also a *, i.e., comments of the form (* * --).

#CAREFULCOLUMNS [Variable]

In the interests of efficiency, PRETTYPRINT approximates the number of characters in each atom, rather than calling NCHARS, when computing how much will fit on a line. This procedure works satisfactorily in most cases. However, users with unusually long atoms in their programs, e.g., such as produced by CLISPIFY, may occasionlly encounter some glitches in the output produced by PRETTYPRINT. The value of #CAREFULCOLUMNS tells PRETTYPRINT how many columns (counting from the right hand margin) in which to actually compute NCHARS instead of approximating. Setting #CAREFULCOLUMNS to 20 or 30 will eliminate the glitches, although it will slow down PRETTYPRINT slightly. #CAREFULCOLUMNS is initially 0.

(**WIDEPAPER** *FLG*) [Function]

(WIDEPAPER T) sets FILELINELENGTH (see Chapter 25), FIRSTCOL, and PRETTYLCOM to large values appropriate for pretty printing files to be listed on wide paper. (WIDEPAPER) restores these parameters to their initial values. WIDEPAPER returns the previous setting of *FLG*.

PRETTYFLG [Variable]

If PRETTYFLG is NIL, PRINTDEF uses PRIN2 instead of prettyprinting. This is useful for producing a fast symbolic dump (see the FAST option of MAKEFILE in Chapter 17). Note that the file loads the same as if it were prettyprinted. PRETTYFLG is initially set to T. PRETTYFLG should not be set to NIL if comment pointers are being used.

CLISPIFYPRETTYFLG [Variable]

Used to inform PRETTYPRINT to call CLISPIFY on selected function definitions before printing them (see Chapter 21).

PRETTYPRINTMACROS [Variable]

An association-list that enables the user to control the formatting of selected expressions. CAR of each expression being PRETTYPRINTed is looked up on PRETTYPRINTMACROS, and if found, CDR of the corresponding entry is applied to the expression. If the result of this application is NIL, PRETTYPRINT ignores the expression; i.e., it prints nothing, assuming that the prettyprintmacro has done any desired printing. If the result of applying the prettyprint macro is non-NIL, the result is prettyprinted in the normal fashion. This gives the user the option of computing some other expression to be prettyprinted in its place.

Note: "prettyprinted in the normal fashion" includes processing prettyprint macros, unless the prettyprint macro returns a structure EQ to the one it was handed, in which case the potential recursion is broken.

PRETTYPRINTYPEMACROS [Variable]

A list of elements of the form (TYPENAME . FN). For types other than lists and atoms, the type name of each datum to be prettyprinted is looked up on

PRETTYPRINTYPEMACROS, and if found, the corresponding function is applied to the datum about to be printed, instead of simply printing it with PRIN2.

PRETTYEQUIVLST [Variable]

An association-list that tells PRETTYPRINT to treat a CAR-of-form the same as some other CAR-of-form. For example, if (QLAMBDA . LAMBDA) appears on PRETTYEQUIVLST, then expressions beginning with QLAMBDA are prettyprinted the same as LAMBDAs. Currently, PRETTYEQUIVLST only allows (i.e., supports in an interesting way) equivalences to forms that PRETTYPRINT internally handles. Equivalence to forms for which the user has specified a prettyprint macro should be made by adding further entries to PRETTYPRINTMACROS

CHANGECHAR [Variable]

If non-NIL, and PRETTYPRINT is printing to a file or display terminal, PRETTYPRINT prints CHANGECHAR in the right hand margin while printing those expressions marked by the editor as having been changed (see Chapter 16). CHANGECHAR is initially |.

26. GRAPHICS OUTPUT OPERATIONS

Streams are used as the basis for all I/O operations. Files are implemented as streams that can support character printing and reading operations, and file pointer manipulation. An image stream is a type of stream that also provides an interface for graphical operations. All of the operations that can applied to streams can be applied to image streams. For example, an image stream can be passed as the argument to PRINT, to print something on an image stream. In addition, special functions are provided to draw lines and curves and perform other graphical operations. Calling these functions on a stream that is not an image stream will generate an error.

Primitive Graphics Concepts

The Interlisp-D graphics system is based on manipulating bitmaps (rectangular arrays of pixels), positions, regions, and textures. These objects are used by all of the graphics functions.

Positions

A position denotes a point in an X,Y coordinate system. A POSITION is an instance of a record with fields XCOORD and YCOORD and is manipulated with the standard record package facilities. For example, (create POSITION XCOORD ← 10 YCOORD ← 20) creates a position representing the point (10,20).

(**POSITIONP** *X*) [Function]

Returns *X* if *X* is a position; NIL otherwise.

Regions

A Region denotes a rectangular area in a coordinate system. Regions are characterized by the coordinates of their bottom left corner and their width and height. A REGION is a record with fields LEFT, BOTTOM, WIDTH, and HEIGHT. It can be manipulated with the standard record package facilities. There are access functions for the REGION record that return the TOP and RIGHT of the region.

The following functions are provided for manipulating regions:

(**CREATEREGION** *LEFT BOTTOM WIDTH HEIGHT*) [Function]

Returns an instance of the REGION record which has *LEFT*, *BOTTOM*, *WIDTH* and *HEIGHT* as respectively its *LEFT*, *BOTTOM*, *WIDTH*, and *HEIGHT* fields.

Example: (CREATEREGION 10 -20 100 200) will create a region that denotes a rectangle whose width is 100, whose height is 200, and whose lower left corner is at the position (10,-20).

(**REGIONP** *X*) [Function]

Returns *X* if *X* is a region, NIL otherwise.

(**INTERSECTREGIONS** $REGION_1$ $REGION_2$ ... $REGION_n$) [NoSpread Function]

Returns a region which is the intersection of a number of regions. Returns NIL if the intersection is empty.

(**UNIONREGIONS** $REGION_1$ $REGION_2$ ... $REGION_n$) [NoSpread Function]

Returns a region which is the union of a number of regions, i.e. the smallest region that contains all of them. Returns NIL if there are no regions given.

(**REGIONSINTERSECTP** $REGION_1$ $REGION_2$) [Function]

Returns T if $REGION_1$ intersects $REGION_2$. Returns NIL if they do not intersect.

(**SUBREGIONP** *LARGEREGION SMALLREGION*) [Function]

Returns T if *SMALLREGION* is a subregion (is equal to or entirely contained in) *LARGEREGION*; otherwise returns NIL.

(**EXTENDREGION** *REGION INCLUDEREGION*) [Function]

Changes (destructively modifies) the region *REGION* so that it includes the region *INCLUDEREGION*. It returns *REGION*.

(**MAKEWITHINREGION** *REGION LIMITREGION*) [Function]

Changes (destructively modifies) the left and bottom of the region *REGION* so that it is within the region *LIMITREGION*, if possible. If the dimension of *REGION* are larger than *LIMITREGION*, *REGION* is moved to the lower left of *LIMITREGION*. If *LIMITREGION* is NIL, the value of the variable *WHOLEDISPLAY* (the screen region) is used. MAKEWITHINREGION returns the modified *REGION*.

(**INSIDEP** *REGION POSORX Y*) [Function]

If *POSORX* and *Y* are numbers, it returns T if the point (*POSORX, Y*) is inside of *REGION*. If *POSORX* is a *POSITION*, it returns T if *POSORX* is inside of *REGION*. If *REGION* is a WINDOW, the window's interior region in window coordinates is used. Otherwise, it returns NIL.

Bitmaps

The display primitives manipulate graphical images in the form of bitmaps. A bitmap is a rectangular array of "pixels," each of which is an integer representing the color of one point in the bitmap image. A bitmap is created with a specific number of bits allocated for each pixel. Most bitmaps used for the display screen use one bit per pixel, so that at most two colors can be represented. If a pixel is 0, the corresponding location on the image is white. If a pixel is 1, its location is black. This interpretation can be changed for the display screen with the function VIDEOCOLOR. Bitmaps with more than one bit per pixel are used to represent color or grey scale images. Bitmaps use a positive integer coordinate system with the lower left corner pixel at coordinate (0,0). Bitmaps are represented as instances of the datatype BITMAP. Bitmaps can be saved on files with the VARS file package command.

(**BITMAPCREATE** *WIDTH HEIGHT BITSPERPIXEL*) [Function]

Creates and returns a new bitmap which is *WIDTH* pixels wide by *HEIGHT* pixels high, with *BITSPERPIXEL* bits per pixel. If BITSPERPIXEL is NIL, it defaults to 1.

(**BITMAPP** *X*) [Function]

Returns *X* if *X* is a bitmap, NIL otherwise.

(**BITMAPWIDTH** *BITMAP*) [Function]

Returns the width of *BITMAP* in pixels.

(**BITMAPHEIGHT** *BITMAP*) [Function]

Returns the height of *BITMAP* in pixels.

(**BITSPERPIXEL** *BITMAP*) [Function]

Returns the number of bits per pixel of *BITMAP*.

(**BITMAPBIT** *BITMAP X Y NEWVALUE*) [Function]

If *NEWVALUE* is between 0 and the maximum value for a pixel in *BITMAP*, the pixel (*X, Y*) is changed to *NEWVALUE* and the old value is returned. If *NEWVALUE* is NIL, *BITMAP* is not changed but the value of the pixel is returned. If *NEWVALUE* is anything else, an error is generated. If (*X, Y*) is outside the limits of *BITMAP*, 0 is returned and no pixels are changed. *BITMAP* can also be a window or display stream. Note: non-window image streams are "write-only"; the *NEWVALUE* argument must be non-NIL.

(**BITMAPCOPY** *BITMAP*) [Function]

Returns a new bitmap which is a copy of *BITMAP* (same dimensions, bits per pixel, and contents).

(**EXPANDBITMAP** *BITMAP WIDTHFACTOR HEIGHTFACTOR*) [Function]

Returns a new bitmap that is *WIDTHFACTOR* times as wide as *BITMAP* a nd *HEIGHTFACTOR* times as high. Each pixel of *BITMAP* is copied into a *WIDTHFACTOR* times *HEIGHTFACTOR* block of pixels. If NIL, *WIDTHFACTOR* defaults to 4, *HEIGHTFACTOR* to 1.

(**ROTATEBITMAP** *BITMAP*) [Function]

Given an m-high by n-wide bitmap, this function returns an n-high by m-wide bitmap. The returned bitmap is the image of the original bitmap, rotated 90 degrees clockwise.

(**SHRINKBITMAP** *BITMAP WIDTHFACTOR HEIGHTFACTOR DESTINATIONBITMAP*) [Function]

Returns a copy of *BITMAP* that has been shrunken by *WIDTHFACTOR* and *HEIGHTFACTOR* in the width and height, respectively. If NIL, *WIDTHFACTOR* defaults to 4, *HEIGHTFACTOR* to 1. If *DESTINATIONBITMAP* is not provided, a bitmap that is 1/*WIDTHFACTOR* by 1/*HEIGHTFACTOR* the size of *BITMAP* is created and returned. *WIDTHFACTOR* and *HEIGHTFACTOR* must be positive integers.

(**PRINTBITMAP** *BITMAP FILE*) [Function]

Prints the bitmap *BITMAP* on the file *FILE* in a format that can be read back in by READBITMAP.

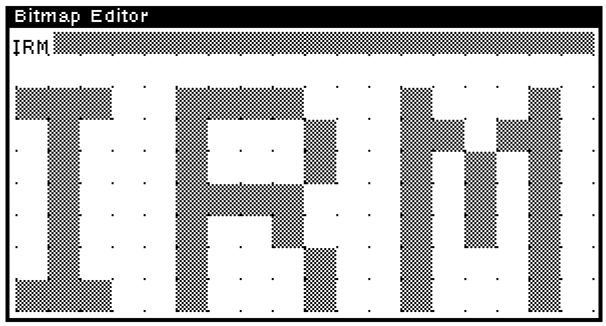
(**READBITMAP** *FILE*) [Function]

Creates a bitmap by reading an expression (written by PRINTBITMAP) from the file *FILE*.

(**EDITBM** *BMSPEC*) [Function]

EDITBM provides an easy-to-use interactive editing facility for various types of bitmaps. If *BMSPEC* is a bitmap, it is edited. If *BMSPEC* is an atom whose value is a bitmap, its value is edited. If *BMSPEC* is NIL, EDITBM asks for dimensions and creates a bitmap. If *BMSPEC* is a region, that portion of the screen bitmap is used. If *BMSPEC* is a window, it is brought to the top and its contents edited.

EDITBM sets up the bitmap being edited in an editing window. The editing window has two major areas: a gridded edit area in the lower part of the window and a display area in the upper left part. In the edit area, the left button will add points, the middle button will erase points. The right button provides access to the normal window commands to reposition and reshape the window. The actual size bitmap is shown in the display area. For example, the following is a picture of the bitmap editing window editing a eight-high by eighteen-wide bitmap:

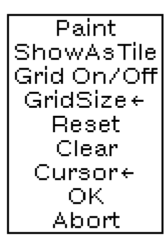


If the bitmap is too large to fit in the edit area, only a portion will be editable. This portion can be changed by scrolling both up and down in the left margin and left and right in the bottom margin. Pressing the middle button while in the display area will bring up a menu that allows global placement of the portion of the bitmap being edited. To allow more of the bitmap to be editing at once, the window can be reshaped to make it larger or the `GridSize←` command described below can be used to reduce the size of a bit in the edit area.

The bitmap editing window can be reshaped to provide more or less room for editing. When this happens, the space allocated to the editing area will be changed to fit in the new region.

Whenever the left or middle button is down and the cursor is not in the edit area, the section of the display of the bitmap that is currently in the edit area is complemented. Pressing the left button while not in the edit region will put the lower left 16 x 16 section of the bitmap into the cursor for as long as the left button is held down.

Pressing the middle button while not in either the edit area or the display area (i.e., while in the grey area in the upper right or in the title) will bring up a command menu.



There are commands to stop editing, to restore the bitmap to its initial state and to clear the bitmap. Holding the middle button down over a command will result in an explanatory message being printed in the prompt window. The commands are described below:

`Paint` Puts the current bitmap into a window and call the window `PAINT` command on it. The `PAINT` command implements drawing with various brush sizes and shapes but only on an actual sized bitmap. The `PAINT` mode is left by pressing the `RIGHT` button and selecting the `QUIT` command from the menu. At this point, you will be given a choice of whether or not the changes you made while in PAINT mode should be made to the current bitmap.

ShowAsTile Tesselates the current bitmap in the upper part of the window. This is useful for determining how a bitmap will look if it were made the display background (using the function CHANGEBACKGROUND). Note: The tiled display will not automatically change as the bitmap changes; to update it, use the ShowAsTile command again.

Grid,On/Off Turns the editing grid display on or off.

GridSize← Allows specification of the size of the editing grid. Another menu will appear giving a choice of several sizes. If one is selected, the editing portion of the bitmap editor will be redrawn using the selected grid size, allowing more or less of the bitmap to be edited without scrolling. The original size is chosen hueristically and is typically about 8. It is particularly useful when editing large bitmaps to set the edit grid size smaller than the original.

Reset Sets all or part of the bitmap to the contents it had when EDITBM was called. Another menu will appear giving a choice between resetting the entire bitmap or just the portion that is in the edit area. The second menu also acts as a confirmation, since not selecting one of the choices on this menu results in no action being taken.

Clear Sets all or part of the bitmap to 0. As with the Reset command, another menu gives a choice between clearing the entire bitmap or just the portion that is in the edit area.

Cursor← Sets the cursor to the lower left part of the bitmap. This prompts the user to specify the cursor "hot spot" by clicking in the lower left corner of the grid.

OK Copies the changed image into the original bitmap, stops the bitmap editor and closes the edit windows. The changes the bitmap editor makes during the interaction occur on a copy of the original bitmap. Unless the bitmap editor is exited via OK, no changes are made in the original.

Stop Stops the bitmap editor without making any changes to the original bitmap.

Textures

A Texture denotes a pattern of gray which can be used to (conceptually) tessellate the plane to form an infinite sheet of gray. It is currently either a 4 by 4 pattern or a 16 by N (N <= 16) pattern. Textures are created from bitmaps using the following function:

(**CREATETEXTUREFROMBITMAP** *BITMAP*) [Function]

Returns a texture object that will produce the texture of *BITMAP*. If *BITMAP* is too large, its lower left portion is used. If *BITMAP* is too small, it is repeated to fill out the texture.

(**TEXTUREP** *OBJECT*) [Function]

Returns *OBJECT* if it is a texture; NIL otherwise.

The functions which accept textures (TEXTUREP, BITBLT, DSPTEXTURE, etc.) also accept bitmaps up to 16 bits wide by 16 bits high as textures. When a region is being filled with a bitmap texture, the texture is treated as if it were 16 bits wide (if less, the rest is filled with white space).

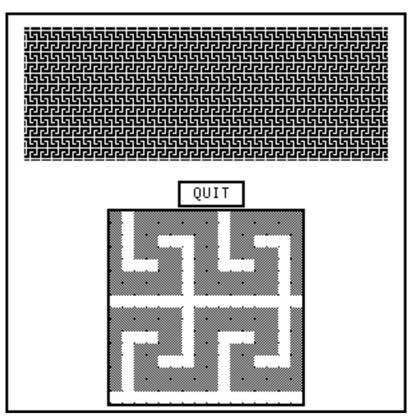
The common textures white and black are available as system constants WHITESHADE and BLACKSHADE. The global variable GRAYSHADE is used by many system facilities as a background gray shade and can be set by the user.

(**EDITSHADE** *SHADE*) [Function]

Opens a window that allows the user to edit textures. Textures can be either small (4 by 4) patterns or large (16 by 16). In the edit area, the left button adds bits to the shade and the middle button erases bits from the shade. The top part of the window is painted with the current texture whenever all mouse keys are released. Thus it is possible to directly compare two textures that differ by more than one pixel by holding a mouse key down until all changes are made. When the "quit" button is selected, the texture being edited is returned.

If *SHADE* is a texture object, EDITSHADE starts with it. If *SHADE* is T, it starts with a large (16 by 16) white texture. Otherwise, it starts with WHITESHADE.

The following is a picture of the texture editor, editing a large (16 by 16) pattern:



Opening Image Streams

An image stream is an output stream which "knows" how to process graphic commands to a graphics output device. Besides accepting the normal character-output functions (PRINT, etc.), an image stream can also be passed as an argument to functions to draw curves, to print characters in multiple fonts, and other graphics operations.

Each image stream has an "image stream type," a litatom that specifies the type of graphic output device that the image stream is processing graphics commands for. Currently, the built-in image stream types are DISPLAY (for the display screen), INTERPRESS (for Interpress format printers), and PRESS (for Press format printers). There are also library packages available that define image stream types for the IRIS display, 4045 printer, FX-80 printer, C150 printer, etc.

Image streams to the display (display streams) interpret graphics commands by immediately executing the appropriate operations to cause the desired image to appear on the display screen. Image streams for hardcopy devices such as Interpress printers interpret the graphic commands by saving information in a file, which can later be sent to the printer.

Note: Not all graphics operations can be properly executed for all image stream types. For example, BITBLT may not be supported to all printers. This functionality is still being developed, but even in the long run some operations may be beyond the physical or logical capabilities of some devices or image file formats. In these cases, the stream will approximate the specified image as best it can.

(**OPENIMAGESTREAM** *FILE IMAGETYPE OPTIONS*) [Function]

Opens and returns an image stream of type *IMAGETYPE* on a destination specified by *FILE*. If *FILE* is a file name on a normal file storage device, the image stream will store graphics commands on the specified file, which can be transmitted to a printer by explicit calls to LISTFILES and SEND.FILE.TO.PRINTER. If *IMAGETYPE* is DISPLAY, then the user is prompted for a window to open. *FILE* in this case will be used as the title of the window.

If *FILE* is a file name on the LPT device, this indicates that the graphics commands should be stored in a temporary file, and automatically sent to the printer when the image stream is closed by CLOSEF. *FILE* = NIL is equivalent to *FILE* = {LPT}. File names on the LPT device are of the form {LPT}PRINTERNAME.TYPE, where PRINTERNAME, TYPE, or both may be omitted. PRINTERNAME is the name of the particular printer to which the file will be transmitted on closing; it defaults to the first printer on DEFAULTPRINTINGHOST that can print *IMAGETYPE* files. The TYPE extension supplies the value of *IMAGETYPE* when it is defaulted (see below). OPENIMAGESTREAM will generate an error if the specified printer does not accept the kind of file specified by *IMAGETYPE*.

If *IMAGETYPE* is NIL, the image type is inferred from the extension field of *FILE* and the EXTENSIONS properties in the list PRINTFILETYPES. Thus, the extensions IP, IPR, and INTERPRESS indicate Interpress format, and the extension PRESS indicates Press format. If *FILE* is a printer file with no extension (of the form {LPT}PRINTERNAME), then *IMAGETYPE* will be the type that the indicated printer can print. If *FILE* has no extension but is not on the printer device {LPT}, then *IMAGETYPE* will default to the type accepted by the first printer on DEFAULTPRINTINGHOST.

OPTIONS is a list in property list format, (PROP1 VAL1 PROP2 VAL2 —), used to specify certain attributes of the image stream; not all attributes are meaningful or interpreted by all types of image streams. Acceptable properties are:

REGION Value is the region on the page (in stream scale units, 0,0 being the lower-left corner of the page) that text will fill up. It establishes the initial values for DSPLEFTMARGIN, DSPRIGHTMARGIN, DSPBOTTOMMARGIN (the point at which carriage returns cause page advancement) and DSPTOPMARGIN (where the stream is positioned at the beginning of a new page).

If this property is not given, the value of the variable DEFAULTPAGEREGION, is used.

FONTS Value is a list of fonts that are expected to be used in the image stream. Some image streams (e.g. Interpress) are more efficient if the expected fonts are specified in advance, but this is not necessary. The first font in this list will be the initial font of the stream, otherwise the default font for that image stream type will be used.

HEADING Value is the heading to be placed automatically on each page. NIL means no heading.

Examples: Suppose that Tremor: is an Interpress printer, Quake is a Press printer, and DEFAULTPRINTINGHOST is (Tremor: Quake):

(OPENIMAGESTREAM) returns an Interpress image stream on printer Tremor:.

(OPENIMAGESTREAM NIL 'PRESS) returns a Press stream on Quake.

(OPENIMAGESTREAM '{LPT}.INTERPRESS) returns an Interpress stream on Tremor:.

(OPENIMAGESTREAM '{CORE}FOO.PRESS) returns a Press stream on the file {CORE}FOO.PRESS.

(**IMAGESTREAMP** *X IMAGETYPE*) [NoSpread Function]

Returns *X* (possibly coerced to a stream) if it is an output image stream of type *IMAGETYPE* (or of any type if *IMAGETYPE* = NIL), otherwise NIL.

(**IMAGESTREAMTYPE** *STREAM*) [Function]

Returns the image stream type of *STREAM*.

(**IMAGESTREAMTYPEP** *STREAM TYPE*) [Function]

Returns T if *STREAM* is an image stream of type *TYPE*.

Accessing Image Stream Fields

The following functions manipulate the fields of an image stream. These functions return the old value (the one being replaced). A value of NIL for the new value will return the current setting without changing it. These functions do not change any of the bits drawn on the image stream; they just affect future operations done on the image stream.

(**DSPCLIPPINGREGION** *REGION STREAM*) [Function]

The clipping region is a region that limits the extent of characters printed and lines drawn (in the image stream's coordinate system). Initially set so that no clipping occurs.

Warning: For display streams, the window system maintains the clipping region during window operations. Users should be very careful about changing this field.

(**DSPFONT** *FONT STREAM*) [Function]

The font field specifies the font used when printing characters to the image stream.

Note: DSPFONT determines its new font descriptor from *FONT* by the same coercion rules that FONTPROP and FONTCREATE use , with one additional possibility: If *FONT* is a list of the form (PROP$_1$ VAL$_1$ PROP$_2$ VAL$_2$ ...) where PROP$_1$ is acceptable as a font-property to FONTCOPY, then the new font is obtained by (FONTCOPY (DSPFONT NIL STREAM) PROP$_1$ VAL$_1$ PROP$_2$ VAL$_2$ ...). For example, (DSPFONT '(SIZE 12) STREAM) would change the font to the 12 point version of the current font, leaving all other font properties the same.

(**DSPTOPMARGIN** *YPOSITION STREAM*) [Function]

The top margin is an integer that is the Y position after a new page (in the image stream's coordinate system). This function has no effect on windows.

(**DSPBOTTOMMARGIN** *YPOSITION STREAM*) [Function]

The bottom margin is an integer that is the minimum Y position that characters will be printed by PRIN1 (in the image stream's coordinate system). This function has no effect on windows.

(**DSPLEFTMARGIN** *XPOSITION STREAM*) [Function]

The left margin is an integer that is the X position after an end-of-line (in the image stream's coordinate system). Initially the left edge of the clipping region.

(**DSPRIGHTMARGIN** *XPOSITION STREAM*) [Function]

The right margin is an integer that is the maximum X position that characters will be printed by PRIN1 (in the image stream's coordinate system). This is initially the position of the right edge of the window or page.

The line length of a window or image stream (as returned by LINELENGTH) is computed by dividing the distance between the left and right margins by the width of an uppercase "A" in the current font. The line length is changed whenever the font, left margin, or right margin are changed or whenever the window is reshaped.

(**DSPOPERATION** *OPERATION STREAM*) [Function]

The operation is the default BITBLT operation used when printing or drawing on the image stream. One of REPLACE, PAINT, INVERT, or ERASE. Initially REPLACE. This is a meaningless operation for most printers which support the model that once dots are deposited on a page they cannot be removed.

(**DSPLINEFEED** *DELTAY STREAM*) [Function]

The linefeed is an integer that specifies the *Y* increment for each linefeed, normally negative. Initially minus the height of the initial font.

(**DSPCLEOL** *DSPSTREAM XPOS YPOS HEIGHT*) [Function]

"Clear to end of line". Clears a region from (*XPOS,YPOS*) to the right margin of the display, with a height of *HEIGHT*. If *XPOS* and *YPOS* are NIL, clears the remainder of the current display line, using the height of the current font.

(**DSPRUBOUTCHAR** *DSPSTREAM CHAR X Y TTBL*) [Function]

Backs up over character code *CHAR* in the *DSPSTREAM*, erasing it. If *X*, *Y* are supplied, the rubbing out starts from the position specified. **DSPRUBOUTCHAR** assumes *CHAR* was printed with the terminal table *TTBL*, so it knows to handle control characters, etc. *TTBL* defaults to the primary terminal table.

(**DSPSCALE** *SCALE STREAM*) [Function]

Returns the scale of the image stream *STREAM*, a number indicating how many units in the streams coordinate system correspond to one printer's point (1/72 of an inch). For example, DSPSCALE returns 1 for display streams, and 35.27778 for Interpress and Press streams (the number of micas per printer's point). In order to be device-independent, user graphics programs must either not specify position values absolutely, or must multiply absolute point quantities by the DSPSCALE of the destination stream. For example, to set the left margin of the Interpress stream XX to one inch, do

```
(DSPLEFTMARGIN (TIMES 72 (DSPSCALE NIL XX)) XX)
```

The SCALE argument to DSPSCALE is currently ignored. In a future release it will enable the scale of the stream to be changed under user control, so that the necessary multiplication will be done internal to the image stream interface. In this case, it would be possible to set the left margin of the Interpress stream XX to one inch by doing

```
(DSPSCALE 1 XX)
(DSPLEFTMARGIN 72 XX)
```

(**DSPSPACEFACTOR** *FACTOR STREAM*) [Function]

The space factor is the amount by which to multiply the natural width of all following space characters on *STREAM*; this can be used for the justification of text. The default value is 1. For example, if the natural width of a space in *STREAM*'s current font is 12 units, and the space factor is set to two, spaces appear 24 units wide. The values returned by STRINGWIDTH and CHARWIDTH are also affected.

The following two functions only have meaning for image streams that can display color:

(**DSPCOLOR** *COLOR STREAM*) [Function]

Sets the default foreground color of *STREAM*. Returns the previous foreground color. If *COLOR* is NIL, it returns the current foreground color without changing anything. The default color is white

(**DSPBACKCOLOR** *COLOR STREAM*) [Function]

Sets the background color of *STREAM*. Returns the previous background color. If *COLOR* is NIL, it returns the current background color without changing anything. The default background color is black.

Current Position of an Image Stream

Each image stream has a "current position," which is a position (in the image stream's coordinate system) where the next printing operation will start from. The functions which print characters or draw on an image stream update these values appropriately. The following functions are used to explicitly access the current position of an image stream:

(**DSPXPOSITION** *XPOSITION STREAM*) [Function]

Returns the X coordinate of the current position of *STREAM*. If *XPOSITION* is non-NIL, the X coordinate is set to it (without changing the Y coordinate).

(**DSPYPOSITION** *YPOSITION STREAM*) [Function]

Returns the Y coordinate of the current position of *STREAM*. If *YPOSITION* is non-NIL, the Y coordinate is set to it (without changing the X coordinate).

(**MOVETO** *X Y STREAM*) [Function]

Changes the current position of *STREAM* to the point (*X*, *Y*).

(**RELMOVETO** *DX DY STREAM*) [Function]

Changes the current position to the point (*DX*, *DY*) coordinates away from current position of *STREAM*.

(**MOVETOUPPERLEFT** *STREAM REGION*) [Function]

Moves the current position to the beginning position of the top line of text. If *REGION* is non-NIL, it must be a *REGION* and the X position is changed to the left edge of *REGION* and the Y position changed to the top of *REGION* less the font ascent of *STREAM*. If *REGION* is NIL, the X coordinate is changed to the left margin of *STREAM* and the Y coordinate is changed to the top of the clipping region of *STREAM* less the font ascent of *STREAM*.

Moving Bits Between Bitmaps With BITBLT

BITBLT is the primitive function for moving bits from one bitmap to another, or from a bitmap to an image stream.

(**BITBLT** *SOURCE SOURCELEFT SOURCEBOTTOM DESTINATION DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION*) [Function]

Transfers a rectangular array of bits from *SOURCE* to *DESTINATION*. *SOURCE* can be a bitmap, or a display stream or window, in which case its associated bitmap is used. *DESTINATION* can be a bitmap or an arbitrary image stream.

WIDTH and *HEIGHT* define a pair of rectangles, one in each of the *SOURCE* and *DESTINATION* whose left, bottom corners are at, respectively, (*SOURCELEFT, SOURCEBOTTOM*) and (*DESTINATIONLEFT, DESTINATIONBOTTOM*). If these rectangles overlap the boundaries of either source or destination they are both reduced in size (without translation) so that they fit within their respective boundaries. If *CLIPPINGREGION* is non-NIL it should be a *REGION* and is interpreted as a clipping region within *DESTINATION*; clipping to this region may further reduce the defining rectangles. These (possibly reduced) rectangles define the source and destination rectangles for BITBLT.

The mode of *transferring* bits is defined by *SOURCETYPE* and *OPERATION*. *SOURCETYPE* and *OPERATION* specify whether the source bits should come from *SOURCE* or *TEXTURE*, and how these bits are combined with those of *DESTINATION*. *SOURCETYPE* and *OPERATION* are described further below.

TEXTURE is a texture. BITBLT aligns the texture so that the upper-left pixel of the texture coincides with the upper-left pixel of the destination bitmap.

SOURCELEFT, SOURCEBOTTOM, DESTINATIONLEFT, and *DESTINATIONBOTTOM* default to 0. *WIDTH* and *HEIGHT* default to the width and height of the *SOURCE*. *TEXTURE* defaults to white. *SOURCETYPE* defaults to INPUT. *OPERATION* defaults to REPLACE. If *CLIPPINGREGION* is not provided, no additional clipping is done. BITBLT returns T if any bits were moved; NIL otherwise.

Note: If *SOURCE* or *DESTINATION* is a window or image stream, the remaining arguments are interpreted as values in the coordinate system of the window or image stream and the operation of BITBLT is translated and clipped accordingly. Also, if a window or image stream is used as the destination to BITBLT, its clipping region further limits the region involved.

SOURCETYPE specifies whether the source bits should come from the bitmap *SOURCE*, or from the texture *TEXTURE*. *SOURCETYPE* is interpreted as follows:

INPUT — The source bits come from *SOURCE*. *TEXTURE* is ignored.

INVERT — The source bits are the inverse of the bits from *SOURCE*. *TEXTURE* is ignored.

TEXTURE — The source bits come from *TEXTURE.* *SOURCE,* *SOURCELEFT*, and *SOURCEBOTTOM* are ignored.

OPERATION specifies how the source bits (as specified by *SOURCETYPE*) are combined with the bits in *DESTINATION* and stored back into *DESTINATION*. *DESTINATION* is one of the following:

REPLACE — All source bits (on or off) replace destination bits.

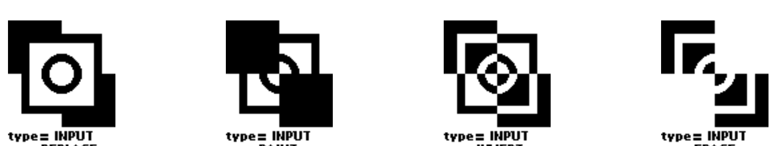
PAINT — Any source bits that are on replace the corresponding destination bits. Source bits that are off have no effect. Does a logical OR between the source bits and the destination bits.

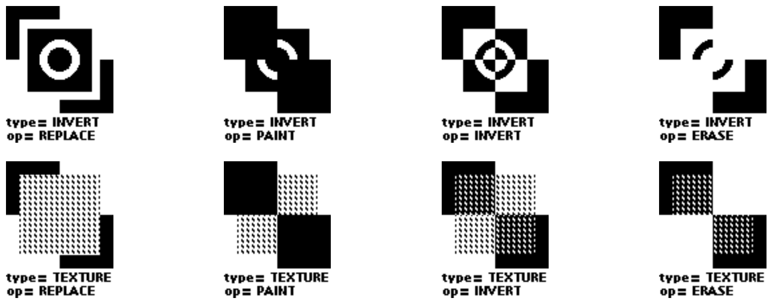
INVERT — Any source bits that are on invert the corresponding destination bits. Does a logical XOR between the source bits and the destination bits.

ERASE — Any source bits that are on erase the corresponding destination bits. Does a logical AND operation between the inverse of the source bits and the destination bits.

Different combinations of *SOURCETYPE* and *OPERATION* can be specified to achieve many different effects. Given the following bitmaps as the values of *SOURCE*, *TEXTURE*, and *DESTINATION*:

BITBLT would produce the results given below for the difference combinations of *SOURCETYPE* and *OPERATION* (assuming *CLIPPINGREGION*, *SOURCELEFT*, etc. are set correctly, of course):





(**BLTSHADE** *TEXTURE DESTINATION DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT OPERATION CLIPPINGREGION*) [Function]

BLTSHADE is the *SOURCETYPE = TEXTURE* case of BITBLT. It fills the specified region of the destination bitmap *DESTINATION* with the texture *TEXTURE*. *DESTINATION* can be a bitmap or image stream.

(**BITMAPIMAGESIZE** *BITMAP DIMENSION STREAM*) [Function]

Returns the size that *BITMAP* will be when BITBLTed to *STREAM*, in *STREAM*'s units. *DIMENSION* can be one of *WIDTH*, *HEIGHT*, or NIL, in which case the dotted pair (*WIDTH . HEIGHT*) will be returned.

Drawing Lines

Interlisp-D provides several functions for drawing lines and curves on image streams. The line drawing functions are intended for interactive applications where efficiency is important. They do not allow the use of "brush" patterns, like the curve drawing functions, but (for display streams) they support drawing a line in INVERT mode, so redrawing the line will erase it. DRAWCURVE can be used to draw lines using a brush.

(**DRAWLINE** X_1 Y_1 X_2 Y_2 *WIDTH OPERATION STREAM COLOR DASHING*) [Function]

Draws a straight line from the point (X_1,Y_1) to the point (X_2,Y_2) on the image stream *STREAM*. The position of *STREAM* is set to (X_2,Y_2). If X_1 equals X_2 and Y_1 equals Y_2, a point is drawn at (X_1,Y_1).

WIDTH is the width of the line, in the units of the device. If *WIDTH* is NIL, the default is 1.

OPERATION is the BITBLT operation used to draw the line. If *OPERATION* is NIL, the value of DSPOPERATION for the image stream is used.

COLOR is a color specification that determines the color used to draw the line for image streams that support color. If *COLOR* is NIL, the *DSPCOLOR* of *STREAM* is used.

DASHING is a list of positive integers that determines the dashing characteristics of the line. The line is drawn for the number of points indicated by the first element of the dashing list, is not drawn for the number of points indicated by the second element. The third element indicates how long it will be on again, and so forth. The dashing sequence is repeated from the beginning when the list is exhausted. A brush LINEWITHBRUSH-by-LINEWITHBRUSH is used.

If *DASHING* is NIL, the line is not dashed.

(**DRAWBETWEEN** *POSITION*$_1$ *POSITION*$_2$ *WIDTH OPERATION STREAM COLOR DASHING*) [Function]

Draws a line from the point *POSITION*$_1$ to the point *POSITION*$_2$ onto the destination bitmap of *STREAM*. The position of *STREAM* is set to *POSITION*$_2$.

In the Medley release, when using the color argument, Interpress **DRAWLINE** treats 16x16 bitmaps or negative numbers as shades/textures. Positive numbers continue to refer to color maps, and so cannot be used as textures. To convert an integer shade into a negative number use NEGSHADE (e.g. (NEGSHADE 42495) is -23041).

(**DRAWTO** *X Y WIDTH OPERATION STREAM COLOR DASHING*) [Function]

Draws a line from the current position to the point (*X,Y*) onto the destination bitmap of *STREAM*. The position of *STREAM* is set to (*X,Y*).

(**RELDRAWTO** *DX DY WIDTH OPERATION STREAM COLOR DASHING*) [Function]

Draws a line from the current position to the point (*DX,DY*) coordinates away onto the destination bitmap of *STREAM*. The position of *STREAM* is set to the end of the line. If *DX* and *DY* are both 0, nothing is drawn.

Drawing Curves

A curve is drawn by placing a brush pattern centered at each point along the curve's trajectory. A brush pattern is defined by its shape, size, and color. The predefined brush shapes are ROUND, SQUARE, HORIZONTAL, VERTICAL, and DIAGONAL; new brush shapes can be created using the INSTALLBRUSH function, described below. A brush size is an integer specifying the width of the brush in the units of the device. The color is a color specification, which is only used if the curve is drawn to an image stream that supports colors.

A brush is specified to the various drawing functions as a list of the form (SHAPE WIDTH COLOR), for example (SQUARE 2) or (VERTICAL 4 RED). A brush can also be specified as a positive integer, which is interpreted as a ROUND brush of that width. If a brush is a litatom, it is assumed to be a function which is called at each point of the curve's trajectory (with three arguments: the X-coordinate of the point, the Y-coordinate, and the image stream), and should do whatever image stream operations are necessary to draw each point. Finally, if a brush is specified as NIL, a (ROUND 1) brush is used as default.

The appearance of a curve is also determined by its dashing characteristics. Dashing is specified by a list of positive integers. If a curve is dashed, the brush is placed along the trajectory for the number of units indicated by the first element of the dashing list. The brush is off, not placed in the bitmap, for a number of units indicated by the second element. The third element indicates how long it will be on again, and so forth. The dashing sequence is repeated from the beginning when the list is exhausted. The units used to measure dashing are the units of the brush. For example, specifying the dashing as (1 1) with a brush of (ROUND 16) would put the brush on the trajectory, skip 16 points, and put down another brush. A curve is not dashed if the dashing argument to the drawing function is NIL.

The curve functions use the image stream's clipping region and operation. Most types of image streams only support the PAINT operation when drawing curves. When drawing to a display stream, the curve-drawing functions accept the operation INVERT if the brush argument is 1. For brushes larger than 1, these functions will use the ERASE operation instead of INVERT. For display streams, the curve-drawing functions treat the REPLACE operation the same as PAINT.

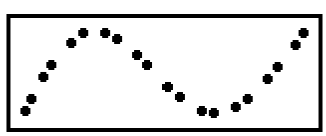
(**DRAWCURVE** *KNOTS CLOSED BRUSH DASHING STREAM*) [Function]

Draws a "parametric cubic spline curve" on the image stream *STREAM*. *KNOTS* is a list of positions to which the curve will be fitted. If *CLOSED* is non-NIL, the curve will be closed; otherwise it ends at the first and last positions in *KNOTS*. *BRUSH* and *DASHING* are interpreted as described above.

For example,

```
(DRAWCURVE '((10 . 10)(50 . 50)(100 . 10)(150 . 50))
   NIL '(ROUND 5) '(1 1 1 2) XX)
```

would draw a curve like the following on the display stream XX:



(**DRAWCIRCLE** *CENTERX CENTERY RADIUS BRUSH DASHING STREAM*) [Function]

Draws a circle of radius *RADIUS* about the point (*CENTERX,CENTERY*) onto the image stream *STREAM*. *STREAM*'s position is left at (*CENTERX,CENTERY*). The other arguments are interpreted as described above.

(**DRAWARC** *CENTERX CENTERY RADIUS STARTANGLE NDEGREES BRUSH DASHINGSTREAM)* [Function]

Draws an arc of the circle whose center point is (*CENTERX CENTERY*) and whose radius is *RADIUS* from the position at *STARTANGLE* degrees for *NDEGREES* number of degrees. If *STARTANGLE* is 0, the starting point will be (*CENTERX* (*CENTERY* + *RADIUS*)). If *NDEGREES* is positive, the arc will be counterclockwise. If *NDEGREES* is negative, the arc will be clockwise. The other arguments are interpreted as described in **DRAWCIRCLE**.

(**DRAWELLIPSE** *CENTERX CENTERY SEMIMINORRADIUS SEMIMAJORRADIUS ORIENTATION BRUSH DASHING*) *STREAM* [Function]

Draws an ellipse with a minor radius of *SEMIMINORRADIUS* and a major radius of *SEMIMAJORRADIUS* about the point (*CENTERX,CENTERY*) onto the image stream *STREAM*. *ORIENTATION* is the angle of the major axis in degrees, positive in the counterclockwise direction. *STREAM*'s position is left at (*CENTERX,CENTERY*). The other arguments are interpreted as described above.

New brush shapes can be defined using the following function:

(**INSTALLBRUSH** *BRUSHNAME BRUSHFN BRUSHARRAY*) [Function]

Installs a new brush called *BRUSHNAME* with creation-function *BRUSHFN* and optional array *BRUSHARRAY*. *BRUSHFN* should be a function of one argument (a width), which returns a bitmap of the brush for that width. *BRUSHFN* will be called to create new instances of *BRUSHNAME*-type brushes; the sixteen smallest instances will be pre-computed and cached. "Hand-crafted" brushes can be supplied as the *BRUSHARRAY* argument. Changing an existing brush can be done by calling INSTALLBRUSH with new *BRUSHFN* and / or *BRUSHARRAY*.

(**DRAWPOINT** *X Y BRUSH STREAM OPERATION*) [Function]

Draws *BRUSH* centered around point (*X, Y*) on *STREAM*, using the operation *OPERATION*. *BRUSH* may be a bitmap or a brush.

Miscellaneous Drawing and Printing Operations

(**DSPFILL** *REGION TEXTURE OPERATION STREAM*) [Function]

Fills *REGION* of the image stream *STREAM* (within the clipping region) with the texture *TEXTURE*. If *REGION* is NIL, the whole clipping region of *STREAM* is used. If *TEXTURE* or *OPERATION* is NIL, the values for *STREAM* are used.

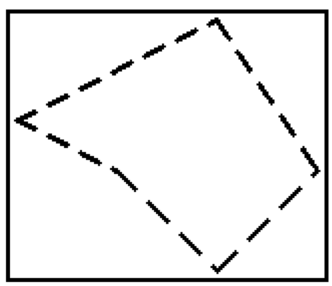
(**DRAWPOLYGON** *POINTS CLOSED BRUSH DASHING STREAM*) [Function]

Draws a polygon on the image stream *STREAM*. *POINTS* is a list of positions to which the figure will be fitted (the vertices of the polygon). If *CLOSED* is non-NIL, then the starting position is specified only once in *POINTS*. If *CLOSED* is NIL, then the starting vertex must be specified twice in *POINTS*. *BRUSH* and *DASHING* are interpreted as described in Chapter 27 of the Interlisp-D Reference Manual.

For example,

```
(DRAWPOLYGON '((100 . 100) (50 . 125)
                 (150 . 175) (200 . 100) (150 .
50))
              T '(ROUND 3) '(4 2) XX)
```

will draw a polygon like the following on the display stream XX.



(**FILLPOLYGON** *POINTS TEXTURE OPERATION WINDNUMBER STREAM*) [Function]

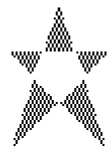
OPERATION is the **BITBLT** operation (see page 27.15 in the Interlisp-D Reference Manual) used to fill the polygon. If the *OPERATION* is NIL, the *OPERATION* defaults to the STREAM default *OPERATION*.

WINDNUMBER is the number for the winding rule convention . This number is either 0 or 1; 0 indicates the "zero" winding rule, 1 indicates the "odd" winding rule.

When filling a polygon, there is more than one way of dealing with the situation where two polygon sides intersect, or one polygon is fully inside the other. Currently, **FILLPOLYGON** to a display stream uses the "odd" winding rule, which means that intersecting polygon sides define areas that are filled or not filled somewhat like a checkerboard. For example,

```
(FILLPOLYGON
  '( ((110 . 110)(150 . 200)(190 . 110))
     ((135 . 125)(160 . 125)(160 . 150)(135 .
150)) )
   GRAYSHADE WINDOW)
```

will produce a display something like this:



This fill convention also takes into account all polygons in *POINTS*, if it specifies multiple polygons.

(**FILLCIRCLE** *CENTERX CENTERY RADIUS TEXTURE STREAM*) [Function]

Fills in a circular area of radius *RADIUS* about the point (*CENTERX, CENTERY*) in *STREAM* with *TEXTURE*. *STREAM*'s position is left at (*CENTERX, CENTERY*).

(**DSPRESET** *STREAM*) [Function]

Sets the X coordinate of *STREAM* to its left margin, sets its Y coordinate to the top of the clipping region minus the font ascent. For a display stream, this also fills its destination bitmap with its background texture.

(**DSPNEWPAGE** *STREAM*) [Function]

Starts a new page. The X coordinate is set to the left margin, and the Y coordinate is set to the top margin plus the linefeed.

(**CENTERPRINTINREGION** *EXP REGION STREAM*) [Function]

Prints *EXP* so that is it centered within *REGION* of the *STREAM*. If *REGION* is NIL, *EXP* will be centered in the clipping region of *STREAM*.

Drawing and Shading Grids

A grid is a partitioning of an arbitrary coordinate system (hereafter referred to as the "source system") into rectangles. This section describes functions that operate on grids. It includes functions to draw the outline of a grid, to translate between positions in a source system and grid coordinates (the coordinates of the rectangle which contains a given position), and to shade grid rectangles. A grid is defined by its "unit grid," a region (called a grid specification) which is the origin rectangle of the grid in terms of the source system. Its LEFT field is interpreted as the X-coordinate of the left edge of the origin rectangle, its BOTTOM field is the Y-coordinate of the bottom edge of the origin rectangle, its WIDTH is the width of the grid rectangles, and its HEIGHT is the height of the grid rectangles.

(**GRID** *GRIDSPEC WIDTH HEIGHT BORDER STREAM GRIDSHADE*) [Function]

Outlines the grid defined by *GRIDSPEC* which is *WIDTH* rectangles wide and *HEIGHT* rectangles high on *STREAM*. Each box in the grid has a border within it that is *BORDER* points on each side; so the resulting lines in the grid are 2**BORDER* thick. If *BORDER* is the atom *POINT*, instead of a border the lower left point of each grid rectangle will be turned on. If *GRIDSHADE* is non-NIL, it should be a texture and the border lines will be drawn using that texture.

(**SHADEGRIDBOX** *X Y SHADE OPERATION GRIDSPEC GRIDBORDER STREAM*) [Function]

Shades the grid rectangle (*X, Y*) of *GRIDSPEC* with texture *SHADE* using *OPERATION* on *STREAM*. *GRIDBORDER* is interpreted the same as for GRID.

The following two functions map from the X,Y coordinates of the source system into the grid X,Y coordinates:

(**GRIDXCOORD** *XCOORD GRIDSPEC*) [Function]

Returns the grid X-coordinate (in the grid specified by *GRIDSPEC*) that contains the source system X-coordinate *XCOORD*.

(**GRIDYCOORD** *YCOORD GRIDSPEC*) [Function]

Returns the grid Y-coordinate (in the grid specified by *GRIDSPEC*) that contains the source system Y-coordinate *YCOORD*.

The following two functions map from the grid X,Y coordinates into the X,Y coordinates of the source system:

(**LEFTOFGRIDCOORD** *GRIDX GRIDSPEC*) [Function]

Returns the source system X-coordinate of the left edge of a grid rectangle at grid X-coordinate *GRIDX* (in the grid specified by *GRIDSPEC*).

(**BOTTOMOFGRIDCOORD** *GRIDY GRIDSPEC*) [Function]

Returns the source system Y-coordinate of the bottom edge of a grid rectangle at grid Y-coordinate *GRIDY* (in the grid specified by *GRIDSPEC*).

Display Streams

Display streams (image streams of type DISPLAY) are used to control graphic output operations to a bitmap, known as the "destination" bitmap of the display stream. For each window on the screen, there is an associated display stream which controls graphics operations to a specific part of the screen bitmap. Any of the functions that take a display stream will also take a window, and use the associated display stream. Display streams can also have a destination bitmap that is not connected to any window or display device.

(**DSPCREATE** *DESTINATION*) [Function]

Creates and returns a display stream. If *DESTINATION* is specified, it is used as the destination bitmap, otherwise the screen bitmap is used.

(**DSPDESTINATION** *DESTINATION DISPLAYSTREAM*) [Function]

Returns the current destination bitmap for *DISPLAYSTREAM*, setting it to *DESTINATION* if non-NIL. *DESTINATION* can be either the screen bitmap, or an auxilliary bitmap in order to construct figures, possibly save them, and then display them in a single operation.

Warning: The window system maintains the destination of a window's display stream. Users should be very careful about changing this field.

(**DSPXOFFSET** *XOFFSET DISPLAYSTREAM*) [Function]

(**DSPYOFFSET** *YOFFSET DISPLAYSTREAM*) [Function]

Each display stream has its own coordinate system, separate from the coordinate system of its destination bitmap. Having the coordinate system local to the display stream allows objects to be displayed at different places by translating the display stream's coordinate system relative to its destination bitmap. This local coordinate system is defined by the X offset and Y offset.

DSPXOFFSET returns the current X offset for *DISPLAYSTREAM*, the X origin of the display stream's coordinate system in the destination bitmap's coordinate system. It is set to *XOFFSET* if non-NIL.

DSPYOFFSET returns the current Y offset for *DISPLAYSTREAM*, the Y origin of the display stream's coordinate system in the destination bitmap's coordinate system. It is set to *YOFFSET* if non-NIL.

The X offset and Y offset for a display stream are both initially 0 (no X or Y-coordinate translation).

Warning: The window system maintains the X and Y offset of a window's display stream. Users should be very careful about changing these fields.

(**DSPTEXTURE** *TEXTURE DISPLAYSTREAM*) [Function]

Returns the current texture used as the background pattern for *DISPLAYSTREAM*. It is set to *TEXTURE* if non-NIL. Initially the value of WHITESHADE.

(**DSPSOURCETYPE** *SOURCETYPE DISPLAYSTREAM*) [Function]

Returns the current BITBLT sourcetype used when printing characters to the display stream. It is set to *SOURCETYPE*, if non-NIL. Must be either INPUT or INVERT. Initially INPUT.

(**DSPSCROLL** *SWITCHSETTING DISPLAYSTREAM*) [Function]

Returns the current value of the "scroll flag," a flag that determines the scrolling behavior of the display stream; either ON or OFF. If ON, the bits in the display streams's destination bitmap are moved after any linefeed that moves the current position out of the destination bitmap. Any bits moved out of the current clipping region are lost. Does not adjust the X offset, Y offset, or clipping region of the display stream. Initially OFF.

Sets the scroll flag to *SWITCHSETTING*, if non-NIL.

Note: The word "scrolling" also describes the use of "scroll bars" on the left and bottom of a window to move an object displayed in a window.

Each window has an associated display stream. To get the window of a particular display stream, use WFROMDS:

(**WFROMDS** *DISPLAYSTREAM DONTCREATE*) [Function]

Returns the window associated with *DISPLAYSTREAM*, creating a window if one does not exist (and *DONTCREATE* is NIL). Returns NIL if the destination of *DISPLAYSTREAM* is not a screen bitmap that supports a window system.

If *DONTCREATE* is non-NIL, WFROMDS will never create a window, and returns NIL if *DISPLAYSTREAM* does not have an associated window.

TTYDISPLAYSTREAM calls WFROMDS with DONTCREATE = T, so it will not create a window unnecessarily. Also, if WFROMDS does create a window, it calls CREATEW with NOOPENFLG = T.

(**DSPBACKUP** *WIDTH DISPLAYSTREAM*) [Function]

Backs up *DISPLAYSTREAM* over a character which is *WIDTH* screen points wide. DSPBACKUP fills the backed over area with the display stream's background texture and decreases the X position by *WIDTH*. If this would put the X position less than *DISPLAYSTREAM*'s left margin, its operation is stopped at the left margin. It returns T if any bits were written, NIL otherwise.

Fonts

A font is the collection of images that are printed or displayed when characters are output to a graphic output device. Some simple displays and printers can only print characters using one font. Bitmap displays and graphic printers can print characters using a large number of fonts.

Fonts are identified by a distinctive style or family (such as Modern or Classic), a size (such as 10 points), and a face (such as bold or italic). Fonts also have a rotation that indicates the orientation of characters on the screen or page. A normal horizontal font (also called a portrait font) has a rotation of 0; the rotation of a vertical (landscape) font is 90 degrees. While any combination can be specified, in practice the user will find that only certain combinations of families, sizes, faces, and rotations are available for any graphic output device.

To specify a font to the functions described below, a FAMILY is represented by a literal atom, a SIZE by a positive integer, and a FACE by a three-element list of the form (WEIGHT SLOPE EXPANSION). WEIGHT, which indicates the thickness of the characters, can be BOLD, MEDIUM, or LIGHT; SLOPE can be ITALIC or REGULAR; and EXPANSION can be REGULAR, COMPRESSED, or EXPANDED, indicating how spread out the characters are. For convenience, faces may also be specified by three-character atoms, where each character is the first letter of the corresponding field. Thus, MRR is a synonym for (MEDIUM REGULAR REGULAR). In addition, certain common face combinations may be indicated by special literal atoms:

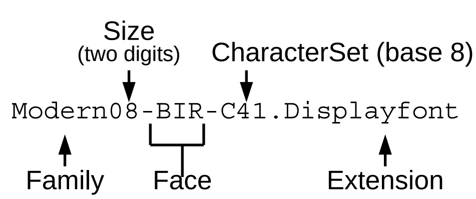
```
STANDARD = (MEDIUM REGULAR REGULAR) = MRR
ITALIC = (MEDIUM ITALIC REGULAR) = MIR
BOLD = (BOLD REGULAR REGULAR) = BRR
BOLDITALIC = (BOLD ITALIC REGULAR) = BIR
```

Interlisp represents all the information related to a font in an object called a font descriptor. Font descriptors contain the family, size, etc. properties used to represent the font. In addition, for each character in the font, the font descriptor contains width information for the character and (for display fonts) a bitmap containing the picture of the character.

The font functions can take fonts specified in a variety of different ways. DSPFONT, FONTCREATE, FONTCOPY, etc. can be applied to font descriptors, "font lists" such as '(MODERN 10), image streams (coerced to its current font), or windows (coerced to the current font of its display stream). The printout command ".FONT" will also accept fonts specified in any of these forms.

In general font files use the following format:

The family name (e.g., Modern); a two digit size (e.g., 08); a three letter Face (e.g., BIR, for Bold Italic Regular); the letter C followed by the font's character set in base 8 (e.g., C41); and finally an extension (e.g., Displayfont).



(**FONTCREATE** *FAMILY SIZE FACE ROTATION DEVICE NOERRORFLG CHARSET*)
[Function]

Returns a font descriptor for the specified font. *FAMILY* is a litatom specifying the font family. *SIZE* is an integer indicating the size of the font in points. *FACE* specifies the face characteristics in one of the formats listed above; if *FACE* is NIL, STANDARD is used. *ROTATION*, which specifies the orientation of the font, is 0 (or NIL) for a portrait font and 90 for a landscape font. *DEVICE* indicates the output device for the font, and can be any image stream type , such as DISPLAY, INTERPRESS, etc. *DEVICE* may also be an image stream, in which case the type of the stream determines the font device. *DEVICE* defaults to DISPLAY.

The *FAMILY* argument to FONTCREATE may also be a list, in which case it is interpreted as a font-specification quintuple, a list of the form (*FAMILY SIZE FACE ROTATION DEVICE*). Thus, (FONTCREATE '(GACHA 10 BOLD)) is equivalent to (FONTCREATE 'GACHA 10 'BOLD). *FAMILY* may also be a font descriptor, in which case that descriptor is simply returned.

If a font descriptor has already been created for the specified font, FONTCREATE simply returns it. If it has not been created, FONTCREATE has to read the font information from a font file that contains the information for that font. The name of an appropriate font file, and the algorithm for searching depends on the device that the font is for, and is described in more detail below. If an appropriate font file is found, it is read into a font descriptor. If no file is found, for DISPLAY fonts FONTCREATE looks for fonts with less face information and fakes the remaining faces (such as by doubling the bit pattern of each character or slanting it). For hardcopy printer fonts, there is no acceptable faking algorithm.

If no acceptable font is found, the action of FONTCREATE is determined by NOERRORFLG. If NOERRORFLG is NIL, it generates a FONT NOT FOUND error with the offending font specification; otherwise, FONTCREATE returns NIL.

CHARSET is the character set which will be read to create the font. Defaults to 0. For more information on character sets, see NS Characters.

(**FONTP** *X*) [Function]

Returns *X* if *X* is a font descriptor; NIL otherwise.

(**FONTPROP** *FONT PROP*) [Function]

Returns the value of the *PROP* property of font *FONT*. The following font properties are recognized:

FAMILY The style of the font, represented as a literal atom, such as CLASSIC or MODERN.

SIZE A positive integer giving the size of the font, in printer's points (1/72 of an inch).

WEIGHT — The thickness of the characters; one of BOLD, MEDIUM, or LIGHT.

SLOPE — The "slope" of the characters in the font; one of ITALIC or REGULAR.

EXPANSION — The extent to which the characters in the font are spread out; one of REGULAR, COMPRESSED, or EXPANDED. Most available fonts have EXPANSION = REGULAR.

FACE — A three-element list of the form (WEIGHT SLOPE EXPANSION), giving all of the typeface parameters.

ROTATION — An integer that gives the orientation of the font characters on the screen or page, in degrees. A normal horizontal font (also called a portrait font) has a rotation of 0; the rotation of a vertical (landscape) font is 90.

DEVICE — The device that the font can be printed on; one of DISPLAY, INTERPRESS, etc.

ASCENT — An integer giving the maximum height of any character in the font from its base line (the printing position). The top line will be at BASELINE+ASCENT-1.

DESCENT — An integer giving the maximum extent of any character below the base line, such as the lower part of a "p". The bottom line of a character will be at BASELINE-DESCENT.

HEIGHT — Equal to ASCENT + DESCENT.

SPEC — The (FAMILY SIZE FACE ROTATION DEVICE) quintuple by which the font is known to Lisp.

DEVICESPEC — The (FAMILY SIZE FACE ROTATION DEVICE) quintuple that identifies what will be used to represent the font on the display or printer. It will differ from the SPEC property only if an implicit coercion is done to approximate the specified font with one that actually exists on the device.

SCALE — The units per printer's point (1/72 of an inch) in which the font is measured. For example, this is 35.27778 (the number of micas per printer's point) for Interpress fonts, which are measured in terms of micas.

(**FONTCOPY** *OLDFONT* $PROP_1$ VAL_1 $PROP_2$ VAL_2 ...) [NoSpread Function]

Returns a font descriptor that is a copy of the font *OLDFONT*, but which differs from *OLDFONT* in that *OLDFONT*'s properties are replaced by the specified properties and values. Thus, (FONTCOPY FONT 'WEIGHT 'BOLD 'DEVICE 'INTERPRESS) will return a bold Interpress font with all other properties the same as those of FONT. FONTCOPY accepts the properties FAMILY, SIZE, WEIGHT, SLOPE, EXPANSION, FACE, ROTATION, and DEVICE. If the first property is a list, it is taken to be the $PROP_1$ VAL_1 $PROP_2$ VAL_2 ... sequence. Thus, (FONTCOPY FONT '(WEIGHT BOLD DEVICE INTERPRESS)) is equivalent to the example above.

If the property NOERROR is specified with value non-NIL, FONTCOPY will return NIL rather than causing an error if the specified font cannot be created.

(**FONTSAVAILABLE** *FAMILY SIZE FACE ROTATION DEVICE CHECKFILESTOO?*) [Function]

Returns a list of available fonts that match the given specification. *FAMILY, SIZE, FACE, ROTATION*, and *DEVICE* are the same as for FONTCREATE. Additionally, any of them can be the atom *, in which case all values of that field are matched.

If *CHECKFILESTOO?* is NIL, only fonts already loaded into virtual memory will be considered. If *CHECKFILESTOO?* is non-NIL, the font directories for the specified device will be searched. When checking font files, the *ROTATION* is ignored.

> Note: The search is conditional on the status of the server which holds the font. Thus a file server crash may prevent FONTCREATE from finding a file that an earlier FONTSAVAILABLE returned.

Each element of the list returned will be of the form (*FAMILY SIZE FACE ROTATION DEVICE*).

Examples:

```
(FONTSAVAILABLE 'MODERN 10 'MRR 0 'DISPLAY)
```

will return ((MODERN 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY)) if the regular Modern 10 font for the display is in virtual memory; NIL otherwise.

```
(FONTSAVAILABLE '* 14 '* '* 'INTERPRESS T)
```

will return a list of all the size 14 Interpress fonts, whether they are in virtual memory or in font files.

(**SETFONTDESCRIPTOR** *FAMILY SIZE FACE ROTATION DEVICE FONT*) [Function]

Indicates to the system that *FONT* is the font that should be associated with the *FAMILY SIZE FACE ROTATION DEVICE* characteristics. If *FONT* is NIL, the font associated with these characteristics is cleared and will be recreated the next time it is needed. As with FONTPROP and FONTCOPY, *FONT* is coerced to a font descriptor if it is not one already.

This functions is useful when it is desirable to simulate an unavailable font or to use a font with characteristics different from the interpretations provided by the system.

(**DEFAULTFONT** *DEVICE FONT* —) [Function]

Returns the font that would be used as the default (if NIL were specified as a font argument) for image stream type *DEVICE*. If *FONT* is a font descriptor, it is set to be the default font for *DEVICE*.

(**CHARWIDTH** *CHARCODE FONT*) [Function]

CHARCODE is an integer that represents a valid character (as returned by CHCON1). Returns the amount by which an image stream's **X**-position will be incremented when the character is printed.

(**CHARWIDTHY** *CHARCODE FONT*) [Function]

Like CHARWIDTH, but returns the **Y** component of the character's width, the amount by which an image stream's **Y**-position will be incremented when the character is printed. This will be zero for most characters in normal portrait fonts, but may be non-zero for landscape fonts or for vector-drawing fonts.

(**STRINGWIDTH** *STR FONT FLG RDTBL*) [Function]

Returns the amount by which a stream's **X**-position will be incremented if the printname for the Interlisp-D object *STR* is printed in font *FONT*. If *FONT* is NIL, DEFAULTFONT is used as *FONT*. If *FONT* is an image stream, its font is used. If *FLG* is non-NIL, the PRIN2-pname of *STR* with respect to the readtable *RDTBL* is used.

(**STRINGREGION** *STR STREAM PRIN2FLG RDTBL*) [Function]

Returns the region occupied by *STR* if it were printed at the current location in the image stream *STREAM*. This is useful, for example, for determining where text is in a window to allow the user to select it. The arguments *PRIN2FLG* and *RDTBL* are passed to STRINGWIDTH.

Note: STRINGREGION does not take into account any carriage returns in the string, or carriage returns that may be automatically printed if *STR* is printed to *STREAM*. Therefore, the value returned is meaningless for multi-line strings.

The following functions allow the user to access and change the bitmaps for individual characters in a display font. Note: Character code 256 can be used to access the "dummy" character, used for characters in the font with no bitmap defined.

(**GETCHARBITMAP** *CHARCODE FONT*) [Function]

Returns a bitmap containing a copy of the image of the character *CHARCODE* in the font *FONT*.

(**PUTCHARBITMAP** *CHARCODE FONT NEWCHARBITMAP NEWCHARDESCENT*) [Function]

Changes the bitmap image of the character *CHARCODE* in the font *FONT* to the bitmap *NEWCHARBITMAP*. If *NEWCHARDESCENT* is non-NIL, the descent of the character is changed to the value of *NEWCHARDESCENT*.

(**EDITCHAR** *CHARCODE FONT*) [Function]

Calls the bitmap editor (EDITBM) on the bitmap image of the character *CHARCODE* in the font *FONT*. *CHARCODE* can be a character code (as returned by CHCON1) or an atom or string, in which case the first character of *CHARCODE* is used.

(**WRITESTRIKEFONTFILE** *FONT CHARSET FILENAME*) [Function]

Takes a display font font descriptor and a character set number, and writes that character set into a file suitable for reading in again. Note that the font descriptor's current state is used (which was perhaps modified by INSPECTing the datum), so this provides a mechanism for creating / modifying new fonts.

For example:

```
(WRITESTRIKEFONTFILE (FONTCREATE 'GACHA 10) 0 '{DSK}Magic10-
MRR-C0.DISPLAYFONT)
```

If your DISPLAYFONTDIRECTORIES includes {DSK}, then a subsequent (FONTCREATE 'MAGIC 10) will create a new font descriptor whose appearance is the same as the old Gacha font descriptor.

However, the new font is identical to the old one in appearance only. The individual datatype fields and bitmap may not be the same as those in the old font descriptor, due to peculiarities of different font file formats.

Font Files and Font Directories

If FONTCREATE is called to create a font that has not been loaded into Interlisp, FONTCREATE has to read the font information from a font file that contains the information for that font. For printer devices, the font files have to contain width information for each character in the font. For display fonts, the font files have to contain, in addition, bitmap images for each character in the fonts. The font file names, formats, and searching algorithms are different for each device. There are a set of variables for each device, that determine the directories that are searched for font files. All of these variables must be set before Interlisp can auto-load font files. These variables should be initialized in the site-specific INIT file.

DISPLAYFONTDIRECTORIES [Variable]

Value is a list of directories searched to find font bitmap files for display fonts.

DISPLAYFONTEXTENSIONS [Variable]

Value is a list of file extensions used when searching DISPLAYFONTDIRECTORIES for display fonts. Initially set to (DISPLAYFONT), but when using older font files it may be necessary to add STRIKE and AC to this list.

INTERPRESSFONTDIRECTORIES [Variable]

Value is a list of directories searched to find font widths files for Interpress fonts.

PRESSFONTWIDTHSFILES [Variable]

Value is a list of files (not directories) searched to find font widths files for Press fonts. Press font widths are packed into large files (usually named FONTS.WIDTHS).

Font Profiles

PRETTYPRINT contains a facility for printing different elements (user functions, system functions, clisp words, comments, etc.) in different fonts to emphasize (or deemphasize) their importance, and in general to provide for a more pleasing appearance. Of course, in order to be useful, this facility requires that the user is printing on a device (such as a bitmapped display or a laser printer) which supports multiple fonts.

PRETTYPRINT signals font changes by inserting into the file a user-defined escape sequence (the value of the variable FONTESCAPECHAR) followed by the character code which specifies, by number, which font to use, i.e. ↑A for font number 1, etc. Thus, if FONTESCAPECHAR were the character ↑F, ↑F↑C would be output to change to font 3, ↑F↑A to change to font 1, etc. If FONTESCAPECHAR consists of characters which are separator charactors in FILERDTBL, then a file with font changes in it can also be loaded back in.

Currently, PRETTYPRINT uses the following font classes. The user can specify separate fonts for each of these classes, or use the same font for several different classes.

LAMBDAFONT The font for printing the name of the function being prettyprinted, before the actual definition (usually a large font).

CLISPFONT If CLISPFLG is on, the font for printing any clisp words, i.e. atoms with property CLISPWORD.

COMMENTFONT The font used for comments.

USERFONT The font for the name of any function in the file, or any member of the list FONTFNS.

SYSTEMFONT The font for any other (defined) function.

CHANGEFONT The font for an expression marked by the editor as having been changed.

PRETTYCOMFONT The font for the operand of a file package command.

DEFAULTFONT The font for everything else.

Note that not all combinations of fonts will be aesthetically pleasing (or even readable!) and the user may have to experiment to find a compatible set.

Although in some implementations LAMBDAFONT et al. may be defined as variables, one should not set them directly, but should indicate what font is to be used for each class by calling the function FONTPROFILE:

(**FONTPROFILE** *PROFILE*) [Function]

Sets up the font classes as determined by *PROFILE*, a list of elements which defines the correspondence between font classes and specific fonts. Each element of *PROFILE* is a list of the form:

(FONTCLASS FONT# DISPLAYFONT PRESSFONT INTERPRESSFONT)

FONTCLASS is the font class name and FONT# is the font number for that class. For each font class name, the escape sequence will consist of FONTESCAPECHAR followed by the character code for the font number, e.g. ↑A for font number 1, etc.

If FONT# is NIL for any font class, the font class named DEFAULTFONT (which must always be specified) is used. Alternatively, if FONT# is the name of a previously defined font class, this font class will be equivalenced to the previously defined one.

DISPLAYFONT, PRESSFONT, and INTERPRESSFONT are font specifications (of the form accepted by FONTCREATE) for the fonts to use when printing to the display and to Press and Interpress printers respectively.

FONTPROFILE [Variable]

This is the variable used to store the current font profile, in the form accepted by the function FONTPROFILE. Note that simply editing this value will not change the fonts used for the various font classes; it is necessary to execute (FONTPROFILE FONTPROFILE) to install the value of this variable.

The process of printing with multiple fonts is affected by a large number of variables: FONTPROFILE, FILELINELENGTH, PRETTYLCOM, etc. To facilitate switching back and forth between various sets of values for the font variables, Interlisp supports the idea of named "font configurations" encapsulating the values of all relevant variables.

To create a new font configuration, set all "relevant" variables to the values you want, and then call FONTNAME to save them (on the variable FONTDEFS) under a given name. To install a particular font configuration, call FONTSET giving it your name. To change the values in a saved font configuration, edit the value of the variable FONTDEFS.

Note: The list of variables saved by FONTNAME is stored in the variable FONTDEFSVARS. This can be changed by the user.

(**FONTSET** *NAME*) [Function]

Installs font configuration for *NAME*. Also evaluates (FONTPROFILE FONTPROFILE) to install the font classes as specified in the new value of the variable FONTPROFILE. Generates an error if *NAME* not previously defined.

FONTDEFSVARS [Variable]

The list of variables to be packaged by a FONTNAME. Initially FONTCHANGEFLG, FILELINELENGTH, COMMENTLINELENGTH, FIRSTCOL, PRETTYLCOM, LISTFILESTR, and FONTPROFILE.

FONTDEFS [Variable]

An association list of font configurations. FONTDEFS is a list of elements of form (NAME . PARAMETER-PAIRS). To save a configuration on a file after performing a FONTNAME to define it, the user could either save the entire value of FONTDEFS, or use the ALISTS file package command to dump out just the one configuration.

FONTESCAPECHAR [Variable]

The character or string used to signal the start of a font escape sequence.

FONTCHANGEFLG [Variable]

If T, enables fonts when prettyprinting. If NIL, disables fonts. ALL indicates that all calls to CHANGEFONT are executed.

LISTFILESTR [Variable]

In Interlisp-10, passed to the operating system by LISTFILES. Can be used to specify subcommands to the LIST command, e.g. to establish correspondance between font number and font name.

COMMENTLINELENGTH [Variable]

Since comments are usually printed in a smaller font, COMMENTLINELENGTH is provided to offset the fact that Interlisp does not know about font widths. When FONTCHANGEFLG = T, CAR of COMMENTLINELENGTH is the linelength used to print short comments, i.e. those printed in the right margin, and CDR is the linelength used when printing full width comments.

(**CHANGEFONT** *FONT STREAM*) [Function]

Executes the operations on *STREAM* to change to the font *FONT*. For use in PRETTYPRINTMACROS.

Image Objects

An Image Object is an object that includes information about an image, such as how to display it, how to print it, and how to manipulate it when it is included in a collection of images (such as a document). More generally, it enables you to include one kind of image, with its own semantics, layout rules, and editing paradigms, inside another kind of image. Image Objects provide a general-purpose interface between image users who want to manipulate arbitrary images, and image producers, who create images for use, say, in documents.

Images are encapsulated inside a uniform barrier—the IMAGEOBJ data type. From the outside, you communicate to the image by calling a standard set of functions. For example, calling one function tells you how big the image is; calling another causes the image object to be displayed where you tell it, and so on. Anyone who wants to create images for general use can implement his own brand of IMAGEOBJ. IMAGEOBJs have been implemented (in library packages) for bitmaps, menus, annotations, graphs, and sketches.

Image Objects were originally implemented to support inserting images into TEdit text files, but the facility is available for use by any tools that manipulate images. The Image Object interface allows objects to exist in TEdit documents and be edited with their own editor. It also provides a facility in which objects can be shift-selected (or "copy-selected") between TEdit and non-TEdit windows. For example, the Image Objects interface allows you to copy-select graphs from a Grapher window into a TEdit window. The source window (where the object comes from) does not have to know what sort of window the destination window (where the object is inserted) is, and the destination does not have to know where the insertion comes from.

A new data type, IMAGEOBJ, contains the data and the procedures necessary to manipulate an object that is to be manipulated in this way. IMAGEOBJs are created with the function IMAGEOBJCREATE (below).

Another new data type, IMAGEFNS, is a vector of the procedures necessary to define the behavior of a type of IMAGEOBJ. Grouping the operations in a separate data type allows multiple instances of the same type of image object to share procedure vectors. The data and procedure fields of an IMAGEOBJ have a uniform interface through the function IMAGEOBJPROP. IMAGEFNS are created with the function IMAGEFNSCREATE:

(**IMAGEFNSCREATE** *DISPLAYFN IMAGEBOXFN PUTFN GETFN COPYFN BUTTONEVENTINFN COPYBUTTONEVENTINFN WHENMOVEDFN WHENINSERTEDFN WHENDELETEDFN WHENCOPIEDFN WHENOPERATEDONFN PREPRINTFN*) — [Function]

Returns an IMAGEFNS object that contains the functions necessary to define the behavior of an IMAGEOBJ.

The arguments *DISPLAYFN* through *PREPRINTFN* should all be function names to be stored as the "methods" of the IMAGEFNS. The purpose of each IMAGEFNS method is described below.

Note: Image objects must be "registered" before they can be read by TEdit or HREAD. IMAGEFNSCREATE implicitly registers its GETFN argument.

(**IMAGEOBJCREATE** *OBJECTDATUM IMAGEFNS*) [Function]

Returns an IMAGEOBJ that contains the object datum *OBJECTDATUM* and the operations vector *IMAGEFNS*. *OBJECTDATUM* can be arbitrary data.

(**IMAGEOBJPROP** *IMAGEOBJECT PROPERTY NEWVALUE*) [NoSpread Function]

Accesses and sets the properties of an IMAGEOBJ. Returns the current value of the *PROPERTY* property of the image object *IMAGEOBJECT*. If *NEWVALUE* is given, the property is set to it.

IMAGEOBJPROP can be used on the system properties OBJECTDATUM, DISPLAYFN, IMAGEBOXFN, PUTFN, GETFN, COPYFN, BUTTONEVENTINFN, COPYBUTTONEVENTINFN, WHENOPERATEDONFN, and PREPRINTFN. Additionally, it can be used to save arbitrary properties on an IMAGEOBJ.

(**IMAGEFNSP** *X*) [Function]

Returns *X* if *X* is an IMAGEFNS object, NIL otherwise.

(**IMAGEOBJP** *X*) [Function]

Returns *X* if *X* is an IMAGEOBJ object, NIL otherwise.

IMAGEFNS Methods

Note: Many of the IMAGEFNS methods below are passed "host stream" arguments. The TEdit text editor passes the "text stream" (an object contain all of the information in the document being edited) as the "host stream" argument. Other editing programs that want to use image objects may want to pass the data structure being edited to the IMAGEFNS methods as the "host stream" argument.

(**DISPLAYFN** *IMAGEOBJ IMAGESTREAM IMAGESTREAMTYPE HOSTSTREAM*) [IMAGEFNS Method]

The DISPLAYFN method is called to display the object *IMAGEOBJ* at the current position on *IMAGESTREAM*. The type of *IMAGESTREAM* indicates whether the device is the display or some other image stream.

Note: When the DISPLAYFN method is called, the offset and clipping regions for the stream are set so the object's image is at (0,0), and only that image area can be modified.

(**IMAGEBOXFN** *IMAGEOBJ IMAGESTREAM CURRENTX RIGHTMARGIN*) [IMAGEFNS Method]

The IMAGEBOXFN method should return the size of the object as an IMAGEBOX, which is a data structure that describes the image laid down when an *IMAGEOBJ* is displayed in terms of width, height, and descender height. An IMAGEBOX has four fields: XSIZE, YSIZE, YDESC, and XKERN. XSIZE and YSIZE are the width and height of the object image. YDESC and XKERN give the position of the baseline and the left edge of the image relative to where you want to position it. For characters, the YDESC is the descent (height of the descender) and the XKERN is the amount of left kerning (note: TEdit doesn't support left kerning).

The IMAGEBOXFN looks at the type of the stream to determine the output device if the object's size changes from device to device. (For example, a bit-map object may specify a scale factor that is ignored when the bit map is displayed on the screen.) *CURRENTX* and *RIGHTMARGIN* allow an object to take account of its environment when deciding how big it is. If these fields are not available, they are NIL.

> Note: TEdit calls the IMAGEBOXFN only during line formatting, then caches the IMAGEBOX as the BOUNDBOX property of the *IMAGEOBJ*. This avoids the need to call the IMAGEBOXFN when incomplete position and margin information is available.

(**PUTFN** *IMAGEOBJ FILESTREAM*) [IMAGEFNS Method]

The PUTFN method is called to save the object on a file. It prints a description on *FILESTREAM* that, when read by the corresponding GETFN method (see below), regenerates the image object. (TEdit and HPRINT take care of writing out the name of the GETFN.)

(**GETFN** *FILESTREAM*) [IMAGEFNS Method]

The GETFN method is called when the object is encountered on the file during input. It reads the description that was written by the PUTFN method and returns an IMAGEOBJ.

(**COPYFN** *IMAGEOBJ SOURCEHOSTSTREAM TARGETHOSTSTREAM*) [IMAGEFNS Method]

The COPYFN method is called during a copy-select operation. It should return a copy of *IMAGEOBJ*. If it returns the litatom DON'T, copying is suppressed.

(**BUTTONEVENTINFN** *IMAGEOBJ WINDOWSTREAM SELECTION RELX RELY WINDOW HOSTSTREAM BUTTON*) [IMAGEFNS Method]

The BUTTONEVENTINFN method is called when you press a mouse button inside the object. The BUTTONEVENTINFN decides whether or not to handle the button, to track the cursor in parallel with mouse movement, and to invoke selections or edits supported by the object (but see the COPYBUTTONEVENTINFN method below). If the BUTTONEVENTINFN returns NIL, TEdit treats the button press as a selection at its level. Note that when this function is first called, a button is down. The BUTTONEVENTINFN should also support the button-down protocol to descend inside of any composite objects with in it. In most cases, the BUTTONEVENTINFN relinquishes control (i.e., returns) when the cursor leaves its object's region.

When the BUTTONEVENTINFN is called, the window's clipping region and offsets have been changed so that the lower-left corner of the object's image is at (0,0), and only the object's image can be changed. The selection is available for changing to fit your needs; the mouse button went down at (RELX,RELY) within the object's image. You can affect how TEdit treats the selection by returning one of several values. If you return NIL, TEdit forgets that you selected an object; if you return the atom DON'T, TEdit doesn't permit the selection; if you return the atom CHANGED, TEdit updates the screen. Use CHANGED to signal TEdit that the object has changed size or will have side effects on other parts of the screen image.

(**COPYBUTTONEVENTINFN** *IMAGEOBJ WINDOWSTREAM*) [IMAGEFNS Method]

The COPYBUTTONEVENTINFN method is called when you button inside an object while holding down a copy key. Many of the comments about BUTTONEVENTINFN apply here too. Also, see the discussion below about copying image objects between windows.

(**WHENMOVEDFN** *IMAGEOBJ TARGETWINDOWSTREAM SOURCEHOSTSTREAM TARGETHOSTSTREAM*) [IMAGEFNS Method]

The WHENMOVEDFN method provides hooks by which the object is notified when TEdit performs an operation (MOVEing) on the whole object. It allows objects to have side effects.

(**WHENINSERTEDFN** *IMAGEOBJ TARGETWINDOWSTREAM SOURCEHOSTSTREAM TARGETHOSTSTREAM*) [IMAGEFNS Method]

The WHENINSERTEDFN method provides hooks by which the object is notified when TEdit performs an operation (INSERTing) on the whole object. It allows objects to have side effects.

(**WHENDELETEDFN** *IMAGEOBJ TARGETWINDOWSTREAM*) [IMAGEFNS Method]

The WHENDELETEDFN method provides hooks by which the object is notified when TEdit performs an operation (DELETEing) on the whole object. It allows objects to have side effects.

(**WHENCOPIEDFN** *IMAGEOBJ TARGETWINDOWSTREAM SOURCEHOSTSTREAM TARGETHOSTSTREAM*) [IMAGEFNS Method]

The WHENCOPIEDFN method provides hooks by which the object is notified when TEdit performs an operation (COPYing) on the whole object. The WHENCOPIEDFN method is called in addition to (and after) the COPYFN method above. It allows objects to have side effects.

(**WHENOPERATEDONFN** *IMAGEOBJ WINDOWSTREAM HOWOPERATEDON SELECTION HOSTSTREAM*) [IMAGEFNS Method]

The WHENOPERATEDONFN method provides a hook for edit operations. HOWOPERATEDON should be one of SELECTED, DESELECTED, HIGHLIGHTED, and UNHILIGHTED. The WHENOPERATEDONFN differs from the BUTTONEVENTINFN because it is called when you extend a selection through the object. That is, the object is treated in toto as a TEdit character. HIGHLIGHTED refers to the selection being highlighted on the screen, and UNHIGHLIGHTED means that the highlighting is being turned off.

(**PREPRINTFN** *IMAGEOBJ*) [IMAGEFNS Method]

The PREPRINTFN method is called to convert the object into something that can be printed for inclusion in documents. It returns an object that the receiving window can print (using either PRIN1 or PRIN2,its choice) to obtain a character representation of the object. If the PREPRINTFN method is NIL, the OBJECTDATUM field of *IMAGEOBJ* itself is used. TEdit uses this function when you indicate that you want to print the characters from an object rather than the object itself (presumably using PRIN1 case).

Registering Image Objects

Each legitimate GETFN needs to be known to the system, to prevent various Trojan-horse problems and to allow the automatic loading of the supporting code for infrequently used IMAGEOBJs. To this end, there is a global list, IMAGEOBJGETFNS, that contains an entry for each GETFN. The existence of the entry marks the GETFN as legitimate; the entry itself is a property list, which can hold information about the GETFN.

No action needs to be taken for GETFNs that are currently in use: the function IMAGEFNSCREATE automatically adds its GETFN argument to the list. However, packages that support obsolete versions of objects may need to explicitly add the obsolete GETFNs. For example, TEdit supports bit-map IMAGEOBJs. Recently, a change was made in the format in which objects are stored; to retain compatibility with the old object format, there are now two GETFNs. The current GETFN is automatically on the list, courtesy of IMAGEFNSCREATE. However, the code file that supports the old bit-map objects contains the clause: (ADDVARS (IMAGEOBJGETFNS (OLDGETFNNAME))), which adds the old GETFN to IMAGEOBJGETFNS.

For a given GETFN, the entry on IMAGEOBJGETFNS may be a property list of information. Currently the only recognized property is FILE.

FILE is the name of the file that can be loaded if the GETFN isn't defined. This file should define the GETFN, along with all the other functions needed to support that kind of IMAGEOBJ.

For example, the bit-map IMAGEOBJ implemented by TEdit use the GETFN BMOBJ.GETFN2. Its entry on IMAGEOBJGETFNS is (BMOBJ.GETFN2 FILE IMAGEOBJ), indicating that the support code for bit-map image objects resides on the file IMAGEOBJ, and that the GETFN for them is BMOBJ.GETFN2.

This makes it possible to have entries for GETFNs whose supporting code isn't loaded—you might, for instance, have your init file add entries to IMAGEOBJGETFNS for the kinds of image objects you commonly use. The system's default reading method will automatically load the code when necessary.

Reading and Writing Image Objects on Files

Image Objects can be written out to files using HPRINT and read back using HREAD. The following functions can also be used:

(**WRITEIMAGEOBJ** *IMAGEOBJ STREAM*) [Function]

Prints (using PRIN2) a call to READIMAGEOBJ, then calls the PUTFN for *IMAGEOBJ* to write it onto *STREAM*. During input, then, the call to READIMAGEOBJ is read and evaluated; it in turn reads back the object's description, using the appropriate GETFN.

(**READIMAGEOBJ** *STREAM GETFN NOERROR*) [Function]

Reads an IMAGEOBJ from *STREAM*, starting at the current file position. Uses the function *GETFN* after validating it (and loading support code, if necessary).

If the *GETFN* can't be validated or isn't defined, READIMAGEOBJ returns an "encapsulated image object", an IMAGEOBJ that safely encapsulates all of the information in the image object. An encapsulated image object displays as a rectangle that says, "Unknown IMAGEOBJ Type" and lists the *GETFN*'s name. Selecting an encapsulated image object with the mouse causes another attempt to read the object from the file; this is so you can load any necessary support code and then get to the object.

Warning: You cannot save an encapsulated image object on a file because there isn't enough information to allow copying the description to the new file from the old one.

If *NOERROR* is non-NIL, READIMAGEOBJ returns NIL if it can't successfully read the object.

Copying Image Objects Between Windows

Copying between windows is implemented as follows: If a button event occurs in a window when a copy key is down, the window's COPYBUTTONEVENTFN window property is called. If this window supports copy-selection, it should track the mouse, indicating the item to be copied. When the button is released, the COPYBUTTONEVENTFN should create an image object out of the selected information, and call COPYINSERT to insert it in the current TTY window. COPYINSERT calls the COPYINSERTFN window property of the TTY window to insert this image object. Therefore, both the source and destination windows can determine how they handle copying image objects.

If the COPYBUTTONEVENTFN of a window is NIL, the BUTTONEVENTFN is called instead when a button event occurs in the window when a copy key is down, and copying from that window is not supported. If the COPYINSERTFN of the TTY window is NIL, COPYINSERT will turn the image object into a string (by calling the PREPRINTFN method of the image object) and insert it by calling BKSYSBUF.

COPYBUTTONEVENTFN [Window Property]

The COPYBUTTONEVENTFN of a window is called (if it exists) when a button event occurs in the window and a copy key is down. If no COPYBUTTONEVENTFN exists, the BUTTONEVENTFN is called.

COPYINSERTFN [Window Property]

The COPYINSERTFN of the "destination" window is called by COPYINSERT to insert something into the destination window. It is called with two arguments: the object to be inserted and the destination window. The object to be inserted can be a character string, an IMAGEOBJ, or a list of IMAGEOBJs and character strings. As a convention, the COPYINSERTFN should call BKSYSBUF if the object to be inserted insert is a character string.

(**COPYINSERT** *IMAGEOBJ*) [Function]

COPYINSERT inserts *IMAGEOBJ* into the window that currently has the TTY. If the current TTY window has a COPYINSERTFN, it is called, passing it *IMAGEOBJ* and the window as arguments.

If no COPYINSERTFN exists and if *IMAGEOBJ* is an image object, BKSYSBUF is called on the result of calling its PREPRINTFN on it. If *IMAGEOBJ* is not an image object, it is simply passed to BKSYSBUF . In this case, BKSYSBUF will call PRIN2 with a read table taken from the process associated with the TTY window. A window that wishes to use PRIN1 or a different read table must provide its own COPYINSERTFN to do this.

Implementation of Image Streams

Interlisp does all image creation through a set of functions and data structures for device-independent graphics, known popularly as DIG. DIG is implemented through the use of a special type of stream, known as an image stream.

An image stream, by convention, is any stream that has its IMAGEOPS field (described in detail below) set to a vector of meaningful graphical operations. Using image streams, you can write programs that draw and print on an output stream without regard to the underlying device, be it a window, a disk, or a printer.

To define a new image stream type, it is necessary to put information on the variable IMAGESTREAMTYPES:

IMAGESTREAMTYPES [Variable]

This variable describes how to create a stream for a given image stream type. The value of IMAGESTREAMTYPES is an association list, indexed by the image stream type (e.g., DISPLAY, INTERPRESS, etc.). The format of a single association list item is:

```
(IMAGETYPE
    (OPENSTREAM OPENSTREAMFN)
    (FONTCREATE FONTCREATEFN)
    (FONTSAVAILABLE FONTSAVAILABLEFN))
```

OPENSTREAMFN, FONTCREATEFN, and FONTSAVAILABLEFN are "image stream methods," device-dependent functions used to implement generic image stream operations. For Interpress image streams, the association list entry is:

```
(INTERPRESS
   (OPENSTREAM OPENIPSTREAM)
   (FONTCREATE \CREATEINTERPRESSFONT)
   (FONTSAVAILABLE \SEARCHINTERPRESSFONTS))
```

(**OPENSTREAMFN** *FILE OPTIONS*) [Image Stream Method]

FILE is the file name as it was passed to OPENIMAGESTREAM, and *OPTIONS* is the *OPTIONS* property list passed to OPENIMAGESTREAM. The result must be a stream of the appropriate image type.

(**FONTCREATEFN** *FAMILY SIZE FACE ROTATION DEVICE*) [Image Stream Method]

FAMILY is the family name for the font, e.g., MODERN. *SIZE* is the body size of the font, in printer's points. *FACE* is a three-element list describing the weight, slope, and expansion of the face desired, e.g., (MEDIUM ITALIC EXPANDED). *ROTATION* is how much the font is to be rotated from the normal orientation, in minutes of arc. For example, to print a landscape page, fonts have the rotation 5400 (90 degrees). The function's result must be a FONTDESCRIPTOR with the fields filled in appropriately.

(**FONTSAVAILABLEFN** *FAMILY SIZE FACE ROTATION DEVICE*) [Image Stream Method]

This function returns a list of all fonts agreeing with the *FAMILY*, *SIZE*, *FACE*, and *ROTATION* arguments; any of them may be wild-carded (i.e., equal to *, which means any value is acceptable). Each element of the list should be a quintuple of the form (*FAMILY SIZE FACE ROTATION DEVICE*).

Where the function looks is an implementation decision: the FONTSAVAILABLEFN for the display device looks at DISPLAYFONTDIRECTORIES, the Interpress code looks on INTERPRESSFONTDIRECTORIES, and implementors of new devices should feel free to introduce new search path variables.

As indicated above, image streams use a field that no other stream uses: IMAGEOPS. IMAGEOPS is an instance of the IMAGEOPS data type and contains a vector of the stream's graphical methods. The methods contained in the IMAGEOPS object can make arbitrary use of the stream's IMAGEDATA field, which is provided for their use, and may contain any data needed.

IMAGETYPE [IMAGEOPS Field]

Value is the name of an image type. Monochrome display streams have an IMAGETYPE of DISPLAY; color display streams are identified as (COLOR DISPLAY). The IMAGETYPE field is informational and can be set to anything you choose.

IMFONTCREATE [IMAGEOPS Field]

Value is the device name to pass to FONTCREATE when fonts are created for the stream.

The remaining fields are all image stream methods, whose value should be a device-dependent function that implements the generic operation. Most methods are called by a similarly-named function, e.g. the function DRAWLINE calls the IMDRAWLINE method. All coordinates that refer to points in a display device's space are measured in the device's units. (The IMSCALE method provides access to a device's scale.) For arguments that have defaults (such as the BRUSH argument of DRAWCURVE), the default is substituted for the NIL argument before it is passed to the image stream method. Therefore, image stream methods do not have to handle defaults.

(**IMCLOSEFN** *STREAM*) [Image Stream Method]

Called before a stream is closed with CLOSEF. This method should flush buffers, write header or trailer information, etc.

(**IMDRAWLINE** *STREAM* X_1 Y_1 X_2 Y_2 *WIDTH OPERATION COLOR DASHING*) [Image Stream Method]

Draws a line of width *WIDTH* from (X_1, Y_1) to (X_2, Y_2). See DRAWLINE.

(**IMDRAWCURVE** *STREAM KNOTS CLOSED BRUSH DASHING*) [Image Stream Method]

Draws a curve through *KNOTS*. See DRAWCURVE.

(**IMDRAWCIRCLE** *STREAM CENTERX CENTERY RADIUS BRUSH DASHING*) [Image Stream Method]

Draws a circle of radius *RADIUS* around (*CENTERX, CENTERY*). See DRAWCIRCLE.

(**IMDRAWELLIPSE** *STREAM CENTERX CENTERY SEMIMINORRADIUS SEMIMAJORRADIUS ORIENTATION BRUSH DASHING*) [Image Stream Method]

Draws an ellipse around (*CENTERX, CENTERY*). See DRAWELLIPSE.

(**IMFILLPOLYGON** *STREAM POINTS TEXTURE*) [Image Stream Method]

Fills in the polygon outlined by *POINTS* on the image stream *STREAM*, using the texture *TEXTURE*. See FILLPOLYGON.

(**IMFILLCIRCLE** *STREAM CENTERX CENTERY RADIUS TEXTURE*) [Image Stream Method]

Draws a circle filled with texture *TEXTURE* around (*CENTERX, CENTERY*). See FILLCIRCLE.

(**IMBLTSHADE** *TEXTURE STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT OPERATION CLIPPINGREGION*) [Image Stream Method]

The texture-source case of BITBLT. *DESTINATIONLEFT*, *DESTINATIONBOTTOM, WIDTH, HEIGHT*, and *CLIPPINGREGION* are measured in *STREAM*'s units. This method is invoked by the functions BITBLT and BLTSHADE.

(**IMBITBLT** *SOURCEBITMAP SOURCELEFT SOURCEBOTTOM STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION CLIPPEDSOURCELEFT CLIPPEDSOURCEBOTTOM SCALE*) [Image Stream Method]

Contains the bit-map-source cases of BITBLT. *SOURCELEFT, SOURCEBOTTOM, CLIPPEDSOURCELEFT, CLIPPEDSOURCEBOTTOM, WIDTH*, and *HEIGHT* are measured in pixels; *DESTINATIONLEFT*, *DESTINATIONBOTTOM*, and *CLIPPINGREGION* are in the units of the destination stream.

(**IMSCALEDBITBLT** *SOURCEBITMAP SOURCELEFT SOURCEBOTTOM STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION CLIPPEDSOURCELEFT CLIPPEDSOURCEBOTTOM SCALE*) [Image Stream Method]

A scaled version of IMBITBLT. Each pixel in *SOURCEBITMAP* is replicated *SCALE* times in the X and Y directions; currently, *SCALE* must be an integer.

(**IMMOVETO** *STREAM X Y*) [Image Stream Method]

Moves to (*X, Y*). This method is invoked by the function MOVETO. If IMMOVETO is not supplied, a default method composed of calls to the IMXPOSITION and IMYPOSITION methods is used.

(**IMSTRINGWIDTH** *STREAM STR RDTBL*) [Image Stream Method]

Returns the width of string *STR* in *STREAM*'s units, using *STREAM*'s current font. This is envoked when STRINGWIDTH is passed a stream as its FONT argument. If IMSTRINGWIDTH is not supplied, it defaults to calling STRINGWIDTH on the default font of *STREAM*.

(**IMCHARWIDTH** *STREAM CHARCODE*) [Image Stream Method]

Returns the width of character *CHARCODE* in *STREAM*'s units, using *STREAM*'s current font. This is invoked when CHARWIDTH is passed a stream as its FONT argument. If IMCHARWIDTH is not supplied, it defaults to calling CHARWIDTH on the default font of *STREAM*.

(**IMCHARWIDTHY** *STREAM CHARCODE*) [Image Stream Method]

Returns the Y componant of the width of character *CHARCODE* in *STREAM*'s units, using *STREAM*'s current font. This is envoked when CHARWIDTHY is passed a stream as its FONT argument. If IMCHARWIDTHY is not supplied, it defaults to calling CHARWIDTHY on the default font of *STREAM*.

(**IMBITMAPSIZE** *STREAM BITMAP DIMENSION*) [Image Stream Method]

Returns the size that *BITMAP* will be when BITBLTed to *STREAM*, in *STREAM*'s units. *DIMENSION* can be one of *WIDTH*, *HEIGHT*, or NIL, in which case the dotted pair (*WIDTH . HEIGHT*) will be returned.

This is envoked by BITMAPIMAGESIZE. If IMBITMAPSIZE is not supplied, it defaults to a method that multiplies the bitmap height and width by the scale of *STREAM*.

(**IMNEWPAGE** *STREAM*) [Image Stream Method]

Causes a new page to be started. The X position is set to the left margin, and the Y position is set to the top margin plus the linefeed. If not supplied, defaults to (\OUTCHAR STREAM (CHARCODE ^L)). Envoked by DSPNEWPAGE.

(**IMTERPRI** *STREAM*) [Image Stream Method]

Causes a new line to be started. The X position is set to the left margin, and the Y position is set to the current Y position plus the linefeed. If not supplied, defaults to (\OUTCHAR STREAM (CHARCODE EOL)). Envoked by TERPRI.

(**IMRESET** *STREAM*) [Image Stream Method]

Resets the X and Y position of *STREAM*. The X coordinate is set to its left margin; the Y coordinate is set to the top of the clipping region minus the font ascent. Envoked by DSPRESET.

The following methods all have corresponding DSPxx functions (e.g., IMYPOSITION corresponds to DSPYPOSITION) that invoke them. They also have the property of returning their previous value; when called with NIL they return the old value without changing it.

(**IMCLIPPINGREGION** *STREAM REGION*) [Image Stream Method]

Sets a new clipping region on *STREAM*.

(**IMXPOSITION** *STREAM XPOSITION*) [Image Stream Method]

Sets the X-position on *STREAM*.

(**IMYPOSITION** *STREAM YPOSITION*) [Image Stream Method]

Sets a new Y-position on *STREAM*.

(**IMFONT** *STREAM FONT*) [Image Stream Method]

Sets *STREAM*'s font to be *FONT*.

(**IMLEFTMARGIN** *STREAM LEFTMARGIN*) [Image Stream Method]

Sets *STREAM*'s left margin to be *LEFTMARGIN*. The left margin is defined as the X-position set after the new line.

(**IMRIGHTMARGIN** *STREAM RIGHTMARGIN*) [Image Stream Method]

Sets *STREAM*'s right margin to be *RIGHTMARGIN*. The right margin is defined as the maximum X-position at which characters are printed; printing beyond it causes a new line.

(**IMTOPMARGIN** *STREAM YPOSITION*) [Image Stream Method]

Sets *STREAM*'s top margin (the Y-position of the tops of characters that is set after a new page) to be *YPOSITION*.

(**IMBOTTOMMARGIN** *STREAM YPOSITION*) [Image Stream Method]

Sets *STREAM*'s bottom margin (the Y-position beyond which any printing causes a new page) to be *YPOSITION*.

(**IMLINEFEED** *STREAM DELTA*) [Image Stream Method]

Sets *STREAM*'s line feed distance (distance to move vertically after a new line) to be *DELTA*.

(**IMSCALE** *STREAM SCALE*) [Image Stream Method]

Returns the number of device points per screen point (a screen point being ~1/72 inch). *SCALE* is ignored.

(**IMSPACEFACTOR** *STREAM FACTOR*) [Image Stream Method]

Sets the amount by which to multiply the natural width of all following space characters on *STREAM*; this can be used for the justification of text. The default value is 1. For example, if the natural width of a space in *STREAM*'s current font is 12 units, and the space factor is set to two, spaces appear 24 units wide. The values returned by STRINGWIDTH and CHARWIDTH are also affected.

(**IMOPERATION** *STREAM OPERATION*) [Image Stream Method]

Sets the default BITBLT *OPERATION* argument.

(**IMBACKCOLOR** *STREAM COLOR*) [Image Stream Method]

Sets the background color of *STREAM*.

(**IMCOLOR** *STREAM COLOR*) [Image Stream Method]

Sets the default color of *STREAM*.

In addition to the IMAGEOPS methods described above, there are two other important methods, which are contained in the stream itself. These fields can be installed using a form like (replace (STREAM OUTCHARFN) of STREAM with (FUNCTION MYOUTCHARFN)). Note: You need to have loaded the Interlisp-D system declarations to manipulate the fields of STREAMs. The declarations can be loaded by loading the Lisp Library package SYSEDIT.

(**STRMBOUTFN** *STREAM CHARCODE*) [Stream Method]

The function called by BOUT.

(**OUTCHARFN** *STREAM CHARCODE*) [Stream Method]

The function that is called to output a single byte. This is like STRMBOUTFN, except for being one level higher: it is intended for text output. Hence, this function should convert (*CHARCODE* EOL) into the stream's actual end-of-line sequence and should adjust the stream's CHARPOSITION appropriately before invoking the stream's STRMBOUTFN (by calling BOUT) to actually put the character. Defaults to \FILEOUTCHARFN, which is probably incorrect for an image stream.

27. WINDOWS AND MENUS

Windows provide a means by which different programs can share a single display harmoniously. Rather than having every program directly manipulating the screen bitmap, all display input/output operations are directed towards windows, which appear as rectangular regions of the screen, with borders and titles. The Interlisp-D window system provides both interactive and programmatic constructs for creating, moving, reshaping, overlapping, and destroying windows in such a way that a program can use a window in a relatively transparent fashion (see the Windows section below). This allows existing Interlisp programs to be used without change, while providing a base for experimentation with more complex windows in new applications.

Menus are a special type of window provided by the window system, used for displaying a set of items to the user, and having the user select one using the mouse and cursor. The window system uses menus to provide the interactive interface for manipulating windows. The menu facility also allows users to create and use menus in interactive programs (see the Menus section below).

Sometimes, a program needs to use a number of windows, displaying related information. The attached window facility (see the Attached Windows section below) makes it easy to manipulate a group of windows as a single unit, moving and reshaping them together.

This chapter documents the Interlisp-D window system. First, it describes the default windows and menus supplied by the window system. Then, the programmatic facilities for creating windows. Next, the functions for using menus. Finally, the attached window facility.

Warning: The window system assumes that all programs follow certain conventions concerning control of the screen. All user programs should use perform display operations using windows and menus. In particular, user programs should not perform operate directly on the screen bitmap; otherwise the window system will not work correctly. For specialized applications that require taking complete control of the display, the window system can be turned off (and back on again) with the following function:

(**WINDOWWORLD** *FLAG*) [NoSpread Function]

The window system is turned on if *FLAG* is `T` and off if *FLAG* is `NIL`. `WINDOWWORLD` returns the previous state of the window system (`T` or `NIL`). If `WINDOWWORLD` is given no arguments, it simply returns the current state without affecting the window system.

Using the Window System

When Medley is initially started, the display screen lights up, showing a number of windows, including the following:

This window is the "logo window," used to identify the system. The logo window is bound to the variable LOGOW until it is closed. The user can create other windows like this by calling the following function:

(**LOGOW** *STRING WHERE TITLE ANGLEDELTA*) [Function]

Creates a window formatted like the "logo window." *STRING* is the string to be printed in big type in the window; if NIL, "Medley" is used. *WHERE* is the position of the lower-left corner of the window; if NIL, the user is asked to specify a position. *TITLE* is the window title to use; if NIL, it defaults to the Xerox copyright notice and date. *ANGLEDELTA* specifies the angle (in degrees) between the boxes in the picture; if NIL, it defaults to 23 degrees.

This window is the "executive window," used for typing expressions and commands to the Interlisp-D executive, and for the executive to print any results (see Chapter 13). For example, in the above picture, the user typed in (PLUS 3 4), the executive evaluated it, and printed out the result, 7. The upward-pointing arrow (**^**) is the flashing caret, which indicates where the next keyboard typein will be printed (see the TTY Process and the Caret section in this chapter).

This window is the "prompt window," used for printing various system prompt messages. It is available to user programs through the following functions:

PROMPTWINDOW [Variable]

Global variable containing the prompt window.

(**PROMPTPRINT** EXP_1 ... EXP_N) [NoSpread Function]

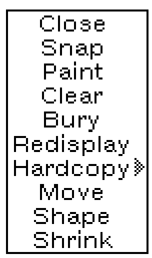
Clears the prompt window, and prints EXP_1 through EXP_N in the prompt window.

(**CLRPROMPT**) [Function]

Clears the prompt window.

The Medley window system allows the user to interactively manipulate the windows on the screen, moving them around, changing their shape, etc. by selecting various operations from a menu.

For most windows, pressing the RIGHT mouse button when the cursor is inside a window during I/O wait will cause the window to come to the top and a menu of window operations to appear.



If a command is selected from this menu (by releasing the right mouse key while the cursor is over a command), the selected operation will be applied to the window in which the menu was brought up. It is possible for an applications program to redefine the action of the RIGHT mouse button. In these cases, there is a convention that the default command menu may be brought up by depressing the RIGHT button when the cursor is in the header or border of a window (see the Mouse Activity in Windows section in this chapter). The operations are:

Close [Window Menu Command]

Closes the window, i.e, removes it from the screen. (See CLOSEW in the Opening and Closing Windows section in this chapter.)

Snap [Window Menu Command]

Prompts for a region on the screen and makes a new window whose bits are a snapshot of the bits currently in that region. Useful for saving some particularly choice image before the window image changes.

Paint [Window Menu Command]

Switches to a mode in which the cursor can be used like a paint brush to draw in a window. This is useful for making notes on a window. While the LEFT button is down, bits are added. While the MIDDLE button is down, they are erased. The RIGHT button pops up a command menu that allows changing of the brush shape, size and shade, changing the mode of combining the brush with the existing bits, or stopping paint mode.

Clear [Window Menu Command]

Clears the window and repositions it to the left margin of the first line of text (below the upper left corner of the window by the amount of the font ascent).

Bury [Window Menu Command]

Puts the window on the bottom of the occlusion stack, thereby exposing any windows that it was hiding.

Redisplay [Window Menu Command]

Redisplays the window. (See REDISPLAYW in the Redisplaying Windows section in this chapter.)

Hardcopy [Window Menu Command]

Prints the contents of the window to the printer. If the window has a window property HARDCOPYFN, it is called with two arguments, the window and an image stream to print to, and the HARDCOPYFN must do the printing. In this way, special windows can be set up that know how to print their contents in a particular way. If the window does not have a HARDCOPYFN, the bitmap image of the window (including the border and title) are printed on the file or printer.

To save the image in a Press or Interpress-format file, or to send it to a non-default printer, use the submenu of the Hardcopy command, indicated by a gray triangle on the right edge of the Hardcopy menu item. If the mouse is moved off of the right of the menu item, another pop-up menu will apear giving the choices "To a file" or "To a printer." If "To a file" is selected, the user is prompted to supply a file name, and the format of the file (Press, Interpress, etc.), and the specified region will be stored in the file.

If "To a printer" is selected, the user is prompted to select a printer from the list of known printers, or to type the name of another printer. If the printer selected is not the first printer on DEFAULTPRINTINGHOST (see Chapter 29), the user will be asked whether to move or add the printer to the beginning of this list, so that future printing will go to the new printer.

Move [Window Menu Command]

Moves the window to a location specified by pressing and then releasing the LEFT button. During this time a ghost frame will indicate where the window will reappear when the key is released. (See GETBOXPOSITION in the Interactive Display Functions section below.)

Shape [Window Menu Command]

Allows the user to specify a new region for the existing window contents. If the LEFT button is used to specify the new region, the reshaped window can be placed anywhere. If the MIDDLE button is used, the cursor will start out tugging at the nearest corner of the existing window, which is useful for making small adjustments in a window that is already positioned correctly. This is done by calling the function SHAPEW (see the Reshaping Windows section below).

Occasionally, a user will have a number of large windows on the screen, making it difficult to access those windows being used. To help with the problem of screen space management, the Interlisp-D window system allows the creation of "icons." An icon is a small rectangle (containing text or a bitmap) which is a "shrunken-down" form of a particular window. Using the Shrink and Expand commands, the user can shrink windows not currently being used into icons, and quickly restore the original windows at any time.

Shrink [Window Menu Command]

Removes the window from the screen and brings up its icon. (See SHRINKW in the Shrinking Windows into Icons section in this chapter) The window can be restored by selecting Expand from the window command menu of the icon.

If the RIGHT button is pressed while the cursor is in an icon, the window command menu will contain a slightly different set of commands. The Redisplay and Clear commands are removed, and the Shrink command is replaced with the Expand command:

Expand [Window Menu Command]

Restores the window associated with this icon and removes the icon. (See EXPANDW in the Shrinking Windows into Icons section in this chapter.)

If the RIGHT button is pressed while the cursor is not in any window, a "background menu" appears with the following operations:

Idle [Background Menu Command]

Enters "idle mode" (see Chapter 12), which blacks out the display screen to save the phosphor. Idle mode can be exited by pressing any key on the keyboard or mouse. This menu command has subitems that allow the user to interactively set idle options to erase the password cache (for security), to request a password before exiting idle mode, to change the timeout before idle mode is entered automatically, etc.

SaveVM [Background Menu Command]

Calls the function SAVEVM (see Chapter 12), which writes out all of the dirty pages of the virtual memory. After a SAVEVM, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the SAVEVM) should you experience a system crash or other disaster.

Snap [Background Menu Command]

The same as the window menu command Snap described above.

Hardcopy [Background Menu Command]

Prompts for a region on the screen, and sends the bitmap image to the printer by calling HARDCOPYW (see Chapter 29). Note that the region can cross window boundaries.

Like the Hardcopy window menu command (above), the user can print to a file or specify a printer by using a submenu.

PSW [Background Menu Command]

Prompts the user for a position on the screen, and creates a "process status window" that allows the user to examine and manipulate all of the existing processes (see Chapter 23).

Various system utilities (TEdit, SEdit, TTYIN) allow information to be "copy-inserted" at the current cursor position by selecting it with the "copy" key held down (Normally the shift keys are the "copy" key; this action can be changed in the key action table.) To "copy-insert" the bitmap of a snap into a Tedit document. If the right mouse button is pressed in the background with the copy key held down, a menu with the single item "SNAP" appears. If this item is selected, the user is prompted to select a region, and a bitmap containing the bits in that region of the screen is inserted into the current tty process, if that process is able to accept image objects.

Some built-in facilities and Lispusers packages add commands to the background menu, to provide an easy way of calling the different facilities. The user can determine what these new commands do by holding the RIGHT button down for a few seconds over the item in question; an explanatory message will be printed in the prompt window.

Changing the Window System

The following functions provide a functional interface to the interactive window operations so that user programs can call them directly.

(**DOWINDOWCOM** *WINDOW*) [Function]

If *WINDOW* is a *WINDOW* that has a DOWINDOWCOMFN window property, it APPLYs that property to *WINDOW*. Shrunken windows have a DOWINDOWCOMFN property that presents a window command menu that contains "expand" instead of "shrink".

If *WINDOW* is a *WINDOW* that doesn't have a DOWINDOWCOMFN window property, it brings up the window command menu. The initial items in these menus are described above. If the user selects one of the items from the provided menu, that item is APPLYed to *WINDOW*.

If *WINDOW* is NIL, DOBACKGROUNDCOM (below) is called.

If *WINDOW* is not a *WINDOW* or NIL, DOWINDOWCOM simply returns without doing anything.

(**DOBACKGROUNDCOM**) [Function]

Brings up the background menu. The initial items in this menu are described above. If the user selects one of the items from the menu, that item is EVALed.

The window command menu for unshrunken windows is cached in the variable WindowMenu. To change the entries in this menu, the user should change the change the menu "command lists" in the variable WindowMenuCommands, and set the appropriate menu variable to a non-MENU, so the menu will be recreated. This provides a way of adding commands to the menu, of changing its font or of restoring the menu if it gets clobbered. The window command menus for icons and the background have similar pairs of variables, documented below. The "command lists" are in the format of the ITEMS field of a menu (see the Menu Fields section below), except as specified below.

Note: Command menus are recreated using the current value of MENUFONT.

WindowMenu [Variable]
WindowMenuCommands [Variable]

The menu that is brought up in response to a right button in an unshrunken window is stored on the variable WindowMenu. If WindowMenu is set to a non-MENU, the menu will be recreated from the list of commands WindowMenuCommands. The CADR of each command added to WindowMenuCommands should be a function name that will be APPLYed to the window.

IconWindowMenu [Variable]
IconWindowMenuCommands [Variable]

The menu that is brought up in response to a right button in a shrunken window is stored on the variable IconWindowMenu. If it is NIL, it is recreated from the list of commands IconWindowMenuCommands. The CADR of each command added a function name that will be APPLYed to the window.

BackgroundMenu [Variable]
BackgroundMenuCommands [Variable]

The menu that is brought up in response to a right button in the background is stored on the variable BackgroundMenu. If it is NIL, it is recreated from the list of commands BackgroundMenuCommands. The CADR of each command added to BackgroundMenuCommands should be a form that will be EVALed.

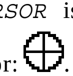
BackgroundCopyMenu [Variable]
BackgroundCopyMenuCommands [Variable]

The menu that is brought up in response to a right button in the background when the copy key is down is stored on the variable BackgroundCopyMenu. If it is NIL, it is recreated from the list of commands BackgroundCopyMenuCommands. The CADR of each command added to BackgroundCopyMenuCommands should be a form that will be EVALed.

Interactive Display Functions

The following functions can be used by programs to allow the user to interactively specify positions or regions on the display screen.

(**GETPOSITION** *WINDOW CURSOR*) [Function]

Returns a POSITION that is specified by the user. GETPOSITION waits for the user to press and release the left button of the mouse and returns the cursor position at the time of release. If *WINDOW* is a *WINDOW*, the position will be in the coordinate system of *WINDOW*'s display stream. If *WINDOW* is NIL, the position will be in screen coordinates. If *CURSOR* is a *CURSOR* (see Chapter 30), the cursor will be changed to it while GETPOSITION is running. If *CURSOR* is NIL, the value of the system variable CROSSHAIRS will be used as the cursor: .

(**GETBOXPOSITION** *BOXWIDTH BOXHEIGHT ORGX ORGY WINDOW PROMPTMSG*) [Function]

Allows the user to position a "ghost" region of size *BOXWIDTH* by *BOXHEIGHT* on the screen, and returns the POSITION of the lower left corner of the region. If *PROMPTMSG* is non-NIL, GETBOXPOSITION first prints it in the PROMPTWINDOW. GETBOXPOSITION then changes the cursor to a box (using the global variable BOXCURSOR:). If *ORGX* and *ORGY* are numbers, they are taken to be the original position of the region, and the cursor is moved to the nearest corner of that region. A ghost region is locked to the cursor so that if the cursor is moved, the ghost region moves with it. If *ORGX* and *ORGY* are numbers, the corner of the region formed by (*ORGX ORGY BOXWIDTH BOXHEIGHT*) that is nearest the cursor position is locked, otherwise the lower left corner is locked. The user can change to another corner by holding down the right button. With the right button down, the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the mouse will snap to the nearest corner, which will then become locked to the cursor. (The held corner can be changed after the left or middle button is down by holding both the original button and the right button down while the cursor is moved to the desired new corner, then letting up just the right button.) When the left or middle button is pressed and released, the lower left corner of the region at the time of release is returned. If *WINDOW* is a *WINDOW*, the returned position will be in *WINDOW*'s coordinate system; otherwise it will be in screen coordinates.

Example:

```
(GETBOXPOSITION 100 200 NIL NIL NIL
     "Specify the position of the command area.")
```

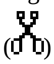
prompts the user for a 100 wide by 200 high region and returns its lower left corner in screen coordinates.

(**GETREGION** *MINWIDTH MINHEIGHT OLDREGION NEWREGIONFN NEWREGIONFNARG INITCORNERS*) [Function]

Lets the user specify a new region and returns that region in screen coordinates. GETREGION prompts for a region by displaying a four-pronged box next to the cursor arrow at one corner of a "ghost" region: . If the user presses the left button, the corner of a "ghost" region opposite the cursor is locked where it is. Once one corner has been fixed, the ghost region expands as the cursor moves.

To specify a region:

1. Move the ghost box so that the corner opposite the cursor is at one corner of the intended region.
2. Press the left button.
3. Move the cursor to the position of the opposite corner of the intended region while holding down the left button.
4. Release the left button.

Before one corner has been fixed, one can switch the cursor to another corner of the ghost region by holding down the right button. With the right button down, the cursor changes to a "forceps" () and the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the cursor will snap to the nearest corner of the ghost region.

After one corner has been fixed, one can still switch to another corner. To change to another corner, continue to hold down the left button and hold down the right button also. With both buttons down, the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the cursor will snap to the nearest corner, which will become the moving corner. In this way, the region may be moved all over the screen, before its size and position is finalized.

The size of the initial ghost region is controlled by the *MINWIDTH*, *MINHEIGHT*, *OLDREGION*, and *INITCORNERS* arguments.

If *INITCORNERS* is non-NIL, it should be a list specifying the initial corners of a ghost region of the form (*BASEX BASEY OPPX OPPY*), where (*BASEX, BASEY*) describes the anchored corner of the box, and (*OPPX, OPPY*) describes the trackable corner (in screen coordinates). The cursor is moved to (*OPPX, OPPY*).

If *INITCORNERS* is NIL, the ghost region will be *MINWIDTH* wide and *MINHEIGHT* high. If *MINWIDTH* or *MINHEIGHT* is NIL, 0 is used. Thus, for a call to GETREGION with no arguments specified, there will be no initial ghost region. The cursor will be in the lower right corner of the region, if there is one.

If *OLDREGION* is a region and the user presses the middle button, the corner of *OLDREGION* farthest from the cursor position is fixed and the corner nearest the cursor is locked to the cursor.

MINWIDTH and *MINHEIGHT*, if given, are the smallest *WIDTH* and *HEIGHT* that the returned region will have. The ghost image will not get any smaller than *MINWIDTH* by *MINHEIGHT*.

If *NEWREGIONFN* is non-NIL, it will be called to determine values for the positions of the corners. This provides a way of "filtering" prospective regions; for instance, by restricting the region to lie on an arbitrary grid. When the user is specifying a region, the region is determined by two of its corners, one that is fixed and one that is tracking the cursor. Each time the cursor moves or a mouse button is pressed, *NEWREGIONFN* is called with three arguments: *FIXEDPOINT*, the position of the fixed corner of the prospective region; *MOVINGPOINT*, the position of the opposite corner of the prospective region; and *NEWREGIONFNARG*. *NEWREGIONFNARG* allows the caller of GETREGION to pass information to the *NEWREGIONFN*.

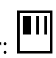
The first time a button is pressed and when the user changes the moving corner via right buttoning, *MOVINGPOINT* is NIL and *FIXEDPOINT* is the position the user selected for the fixed corner of the new region. In this case, the position returned by *NEWREGIONFN* will be used for the fixed corner instead of the one proposed by the user. For all other calls, *FIXEDPOINT* is the position of the fixed corner (as returned by the previous call) and

MOVINGPOINT is the new position the user selected for the opposite corner. In these cases, the value of *NEWREGIONFN* is used for the opposite corner instead of the one proposed by the user. In all cases, the ghost region is drawn with the values returned by *NEWREGIONFN*. *NEWREGIONFN* can be a list of functions in which case they are called in order with each being passed the result of calling the previous and the value of the last one used as the point.

(**GETBOXREGION** *WIDTH HEIGHT ORGX ORGY WINDOW PROMPTMSG*) [Function]

Performs the same prompting as GETBOXPOSITION and returns the REGION specified by the user instead of the POSITION of its lower left corner.

(**MOUSECONFIRM** *PROMPTSTRING HELPSTRING WINDOW DON'TCLEARWINDOWFLG*) [Function]

MOUSECONFIRM provides a simple way for the user to confirm or abort some action simply by using the mouse buttons. It prints the strings *PROMPTSTRING* and *HELPSTRING* in the window *WINDOW*, changes the cursor to a "little mouse" cursor:  (stored in the variable MOUSECONFIRMCURSOR), and waits for the user to press the left button to confirm, or any other button to abort. If the left button was the last button released, returns T, else NIL.

If *PROMPTSTRING* is NIL, it is not printed out. If *HELPSTRING* is NIL, the string "Click LEFT to confirm, RIGHT to abort." is used. If *WINDOW* is NIL, the prompt window is used.

Normally, MOUSECONFIRM clears *WINDOW* before returning. If *DON'TCLEARWINDOWFLG* is non-NIL, the window is not cleared.

Windows

A window specifies a region of the screen, a display stream, functions that get called when the window undergoes certain actions, and various other items of information. The basic model is that a window is a passive collection of bits (on the screen). On top of this basic level, the system supports many different types of windows that are linked to the data structures displayed in them and provide selection and redisplaying routines. In addition, it is possible for the user to create new types of windows by providing selection and displaying functions for them.

Windows are ordered in depth from user to background. Windows in front of others obscure the latter. Operating on a window generally brings it to the top.

Windows are located at a certain position on the screen. Each window has a clipping region that confines all bits written to it to a region that allows a border around the window, and a title above it.

Each window has a display stream associated with it (see Chapter 27), and either a window or its display stream can be passed interchangeably to all system functions. There are dependencies between the window and its display stream that the user should not disturb. For instance, the destination bitmap of the display stream of a window must always be the screen bitmap. The X offset, Y offset, and Clipping Region fields of the display stream should not be changed.

Windows can be created by the user interactively, under program control, or may be created automatically by the system.

Windows are in one of two states: "open" or "closed". In an "open" state, a window is visible on the screen (unless it is covered by other open windows or off the edge of the screen) and accessible to mouse operations. In a "closed" state, a window is not visible and not accessible to mouse operations. Any attempt to print or draw on a closed window will open it.

Window Properties

The behavior of a window is controlled by a set of "window properties." Some of these are used by the system. However, any arbitrary property name may be used by a user program to associate information with a window. For many applications the user will associate the structure being displayed with its window using a property. The following functions provide for reading and setting window properties:

(**WINDOWPROP** *WINDOW PROP NEWVALUE*) [NoSpread Function]

Returns the previous value of *WINDOW*'s *PROP* aspect. If *NEWVALUE* is given, (even if given as NIL), it is stored as the new *PROP* aspect. Some aspects cannot be set by the user and will generate errors. Any *PROP* name that is not recognized is stored on a property list associated with the window.

(**WINDOWADDPROP** *WINDOW PROP ITEMTOADD FIRSTFLG*) [Function]

WINDOWADDPROP adds a new item to a window property. If *ITEMTOADD* is EQ to an element of the *PROP* property of the window WINDOW, nothing is added. If the current property is not a list, it is made a list before *ITEMTOADD* added. WINDOWADDPROP returns the previous property. If *FIRSTFLG* is non-NIL, the new item goes on the front of the list; otherwise, it goes on the end of the list. If *FIRSTFLG* is non-NIL and *ITEMTOADD* is already on the list, it is moved to the front.

Many window properties (*OPENFN, CLOSEFN*, etc.) can be a list of functions. WINDOWADDPROP is useful for adding additional functions to a window property without affecting any existing functions. Note that if the order of items in a window property is important, the list can be modified using WINDOWPROP.

(**WINDOWDELPROP** *WINDOW PROP ITEMTODELETE*) [Function]

WINDOWDELPROP deletes *ITEMTODELETE* from the window property *PROP* of *WINDOW* and returns the previous list if *ITEMTODELETE* was an element. If *ITEMTODELETE* was not a member of window property *PROP*, NIL is returned.

Creating Windows

(**CREATEW** *REGION TITLE BORDERSIZE NOOPENFLG*) [Function]

Creates a new window. *REGION* indicates where and how large the window should be by specifying the exterior region of the window. The usable height and width of the resulting window will be smaller than the height and width of the region by twice the border size and further less the height of the title, if any. If *REGION* is NIL, GETREGION is called to prompt the user for a region.

If *TITLE* is non-NIL, it is printed in the border at the top of the window. The *TITLE* is printed using the global display stream WindowTitleDisplayStream. Thus the height of the title will be (FONTPROP WindowTitleDisplayStream 'HEIGHT).

If *BORDERSIZE* is a number, it is used as the border size. If *BORDERSIZE* is not a number, the window will have a border WBorder (initially 4) bits wide.

If *NOOPENFLG* is non-NIL, the window will not be opened, i.e. displayed on the screen.

The initial X and Y positions of the window are set to the upper left corner by calling MOVETOUPPERLEFT (see Chapter 27).

(**DECODE.WINDOW.ARG** *WHERESPEC WIDTH HEIGHT TITLE BORDER NOOPENFLG*) [Function]

This is a useful function for creating windows. *WHERESPEC* can be a WINDOW, a REGION, a POSITION or NIL. If *WHERESPEC* is a WINDOW, it is returned. In all other cases, CREATEW is called with the arguments *TITLE BORDER* and *NOOPENFLG*. The REGION argument to CREATEW is determined from *WHERESPEC* as follows:

If *WHERESPEC* is a REGION, it is adjusted to be on the screen, then passed to CREATEW.

If *WIDTH* and *HEIGHT* are numbers and *WHERESPEC* is a POSITION, the region whose lower left corner is *WHERESPEC*, whose width is *WIDTH* and whose height is *HEIGHT* is adjusted to be on the screen, then passed to CREATEW.

If *WIDTH* and *HEIGHT* are numbers and *WHERESPEC* is not a POSITION, then GETBOXREGION is called to prompt the user for the position of a region that is WIDTH by *HEIGHT*.

If *WIDTH* and *HEIGHT* are not numbers, CREATEW is given NIL as a REGION argument.

If *WIDTH* and *HEIGHT* are used, they are used as interior dimensions for the window.

(**WINDOWP** *X*) [Function]

Returns *X* if *X* is a window, NIL otherwise.

Opening and Closing Windows

(**OPENWP** *WINDOW*) [Function]

Returns *WINDOW*, if *WINDOW* is an open window (has not been closed); NIL otherwise.

(**OPENWINDOWS**) [Function]

Returns a list of all open windows.

(**OPENW** *WINDOW*) [Function]

If *WINDOW* is a closed window, OPENW calls the function or functions on the window property OPENFN of *WINDOW*, if any. If one of the OPENFNs is the atom DON'T, the window will not be opened. Otherwise the window is placed on the occlusion stack of windows and its contents displayed on the screen. If *WINDOW* is an open window, it returns NIL.

(**CLOSEW** *WINDOW*) [Function]

CLOSEW calls the function or functions on the window property CLOSEFN of *WINDOW*, if any. If one of the CLOSEFNs is the atom DON'T or returns the atom DON'T as a value, CLOSEW returns without doing anything further. Otherwise, CLOSEW removes *WINDOW* from the window stack and restores the bits it is obscuring. If *WINDOW* was closed, *WINDOW* is returned as the value. If it was not closed, (for example because its CLOSEFN returned the atom DON'T), NIL is returned as the value.

WINDOW can be restored in the same place with the same contents (reopened) by calling OPENW or by using it as the source of a display operation.

OPENFN [Window Property]

The OPENFN window property can be a single function or a list of functions. If one of the OPENFNs is the atom DON'T, the window will not be opened. Otherwise, the OPENFNs are called after a window has been opened by OPENW, with the window as a single argument.

CLOSEFN [Window Property]

The CLOSEFN window property can be a single function or a list of functions that are called just before a window is closed by CLOSEW. The function(s) will be called with the window as a single argument. If any of the CLOSEFNs are the atom DON'T, or if the value returned by any of the CLOSEFNs is the atom DON'T, the window will not be closed.

Note: If the CAR of the CLOSEFN list is a LAMBDA word, it is treated as a single function.

Note: A CLOSEFN should not call CLOSEW on its argument.

Redisplaying Windows

(**REDISPLAYW** *WINDOW REGION ALWAYSFLG*) [Function]

Redisplay the region *REGION* of the window *WINDOW*. If *REGION* is NIL, the entire window is redisplayed.

If *WINDOW* doesn't have a REPAINTFN, the action depends on the value of *ALWAYSFLG*. If *ALWAYSFLG* is NIL, *WINDOW* will not change and the message "Window has no REPAINTFN. Can't redisplay." will be printed in the prompt window. If *ALWAYSFLG* is non-NIL, REDISPLAYW acts as if REPAINTFN was NILL.

REPAINTFN [Window Property]

The REPAINTFN window property can be a single function or a list of functions that are called to repaint parts of the window by REDISPLAYW. The REPAINTFNs are called with two arguments: the window and the region in the coordinates of the window's display stream of the area that should be repainted. Before the REPAINTFN is called, the clipping region of the window is set to clip all display operations to the area of interest so that the REPAINTFN can display the entire window contents and the results will be appropriately clipped.

Note: CLEARW (see the Miscellaneous Window Functions section below) should not be used in REPAINTFNs because it resets the window's coordinate system. If a REPAINTFN wants to clear its region first, it should use DSPFILL (see Chapter 27).

Reshaping Windows

(**SHAPEW** *WINDOW NEWREGION*) [Function]

Reshapes *WINDOW*. If the window property *RESHAPEFN* is the atom DON'T or a list that contains the atom DON'T, a message is printed in the prompt window, *WINDOW* is not changed, and NIL is returned. Otherwise, RESHAPEFN window property can be a single function or a list of functions that are called when a window is reshaped, to reformat or redisplay the window contents (see below). If the RESHAPEFN window property is NIL, RESHAPEBYREPAINTFN is the default.

If the region *NEWREGION* is NIL, it prompts for a region with GETREGION. When calling GETREGION, the function MINIMUMWINDOWSIZE is called to determine the minimum height and width of the window, the function WINDOWREGION is called to get the region passed as the OLDREGION argument, the window property NEWREGIONFN is used as the NEWREGIONFN argument and *WINDOW* as the NEWREGIONFNARG argument. If the window property INITCORNERSFN is non-NIL, it is applied to the window, and the value is passed as the INITCORNERS argument to GETREGION, to determine the initial size of the "ghost region." These window properties allow the window to specify the regions used for interactive calls to SHAPEW.

If the region *NEWREGION* is a REGION and its WIDTH or HEIGHT less than the minimums returned by calling the function MINIMUMWINDOWSIZE, they will be increased to the minimums.

If *WINDOW* has a window property DOSHAPEFN, it is called, passing it *WINDOW* and NEWREGION (or the region returned by GETREGION). If *WINDOW* does not have a DOSHAPEFN window property, the function SHAPEW1 is called to reshape the window.

DOSHAPEFNs are provided to implement window groups and few users should ever write them. They are tricky to write and must call SHAPEW1 eventually. The RESHAPEFN window property is a simpler hook into reshape operations.

(**SHAPEW1** *WINDOW REGION*) [Function]

Changes *WINDOW*'s size and position on the screen to be *REGION*. After clearing the region on the screen, it calls the window's RESHAPEFN, if any, passing it three arguments: *WINDOW*; a bitmap that contains *WINDOW*'s previous screen image; and the region of *WINDOW*'s old image within the bitmap.

RESHAPEFN [Window Property]

The RESHAPEFN window property can be a single function or a list of functions that are called when a window is reshaped by SHAPEW. If the RESHAPEFN is DON'T or a list containing DON'T, the window will not be reshaped. Otherwise, the function(s) are called after the window has been reshaped, its coordinate system readjusted to the new position, the title and border displayed, and the interior filled with texture. The RESHAPEFN should display any additional information needed to complete the window's image in the new position and shape. The RESHAPEFN is called with four arguments: (1) the window in its reshaped form, (2) a bitmap with the image of the old window in its old shape, and (3) the region within the bitmap that contains the window's old image, and (4) the region of the screen previously occupied by this window. This function is provided so that users can reformat window contents or whatever. RESHAPEBYREPAINTFN (below) is the default and should be useful for many windows.

NEWREGIONFN [Window Property]

If SHAPEW calls GETREGION to prompt the user for a region, the value of the NEWREGIONFN window property is passed as the NEWREGIONFN argument to GETREGION.

INITCORNERSFN [Window Property]

If this window property is non-NIL, it should be a function of one argument, a window, that returns a list specifying the initial corners of a "ghost region" of the form (BASEX BASEY OPPX OPPY), where (BASEX, BASEY) describes the anchored corner of the box, and (OPPX, OPPY) describes the trackable corner. If SHAPEW calls GETREGION to prompt the user for a region, this function is applied to the window, and the list returned is passed as the INITCORNERS argument to GETREGION, to specify the initial ghost region.

DOSHAPEFN [Window Property]

If this window property is non-NIL, it is called by SHAPEW to reshape the window (instead of SHAPEW1). It is called with two arguments: the window and the new region.

(**RESHAPEBYREPAINTFN** *WINDOW OLDIMAGE IMAGEREGION OLDSCREENREGION*)
[Function]

This the default window RESHAPEFN. *WINDOW* is a window that has been reshaped from the screen region *OLDSCREENREGION* to its new region (available via (WINDOWPROP WINDOW 'REGION)). *OLDIMAGE* is a bitmap that contains the image of the window from its previous location. *IMAGEREGION* is the region within *OLDIMAGE* that contains the old image.

RESHAPEBYREPAINTFN BITBLTs the old region contents into the new region. If the new shape is larger in either or both dimensions, the newly exposed areas are redisplayed via calls *WINDOW*'s REPAINTFN window property. RESHAPEBYREPAINTFN may call the REPAINTFN up to four times during a single reshape.

The choice of which areas of the window to remove or extend is done as follows. If *WINDOW*'s new region shares an edge with OLDSCREENREGION, that edge of the window image will remain fixed and any addition or reduction in that dimension will be performed on the opposite side. If *WINDOW* has an EXTENT property and the newly exposed window area is outside of it, any extra will be added so as to show EXTENT that was previously not visible. An exception to these rules is that the current X,Y position is kept visible, if it was visible before the reshape.

Moving Windows

(**MOVEW** *WINDOW POSorX Y*) [Function]

Moves *WINDOW* to the position specified by *POSorX* and *Y* according to the following rules:

If *POSorX* is NIL, GETBOXPOSITION is called to read a position from the user. If *WINDOW* has a CALCULATEREGION window property, it will be called with *WINDOW* as an argument and should return a region which will be used to prompt the user with. If *WINDOW* does not have a CALCULATEREGION window property, the region of *WINDOW* is used to prompt with.

If *POSorX* is a POSITION, *POSorX* is used.

If *POSorX* and *Y* are both NUMBERP, a position is created using *POSorX* as the XCOORD and *Y* as the YCOORD.

If *POSorX* is a REGION, a position is created using its LEFT as the XCOORD and BOTTOM as the YCOORD.

If *WINDOW* is not open and *POSorX* is non-NIL, the window will be moved without being opened. Otherwise, it will be opened.

If *WINDOW* has the atom DON'T as a MOVEFN window property, the window will not be moved. If *WINDOW* has any other non-NIL value as a MOVEFN property, it should be a function or list of functions that will be called before the window is moved with the *WINDOW* and the new positon as its arguments. If it returns the atom DON'T, the window will not be moved. If it returns a position, the window will be moved to that position instead of the new one. If there are more than one MOVEFNs, the last one to return a value is the one that determines where the window is moved to.

If *WINDOW* is moved and *WINDOW* has an AFTERMOVEFN window property, it should be a function or a list of functions that will be called after the window is moved with *WINDOW* as an argument.

MOVEW returns the new position, or NIL if the window could not be moved.

Note: If MOVEW moves any part of the window from off-screen onto the screen, that part is redisplayed (by calling REDISPLAYW).

(**RELMOVEW** *WINDOW POSITION*) [Function]

Like MOVEW for moving windows but the *POSITION* is interpreted relative to the current position of *WINDOW*. Example: The following code moves *WINDOW* to the right one screen point.

```
(RELMOVEW WINDOW (create POSITION XCOORD ← 1 YCOORD
← 0))
```

CALCULATEREGION [Window Property]

If MOVEW calls GETBOXPOSITION to prompt the user for a region, the CALCULATEREGION window property is called (passing the window as an argument. The CALCULATEREGION should returns a region to be used to prompt the user with. If CALCULATEREGION is NIL, the region of the window is used to prompt with.

MOVEFN [Window Property]

If the MOVEFN is DON'T, the window will not be moved by MOVEW. Otherwise, if the MOVEFN is non-NIL, it should be a function or a list of functions that will be called before a window is moved with two arguments: the window being moved and the new position of the lower left corner in screen coordinates. If the MOVEFN returns DON'T, the window will not be moved. If the MOVEFN returns a POSITION, the window will be moved to that position. Otherwise, the window will be moved to the specified new position.

AFTERMOVEFN [Window Property]

If non-NIL, it should be a function or a list of functions that will be called after the window is moved (by MOVEW) with the window as an argument.

Exposing and Burying Windows

(**TOTOPW** *WINDOW NOCALLTOTOPFNFLG*) [Function]

Brings *WINDOW* to the top of the stack of overlapping windows, guaranteeing that it is entirely visible. If *WINDOW* is closed, it is opened. This is done automatically whenever a printing or drawing operation occurs to the window.

If NOCALLTOTOPFNFLG is NIL, the TOTOPFN of *WINDOW* is called. If *NOCALLTOTOPFNFLG* is T, it is not called, which allows a TOTOPFN to call TOTOPW without causing an infinite loop.

(**BURYW** *WINDOW*) [Function]

Puts *WINDOW* on the bottom of the stack by moving all the windows that it covers in front of it.

TOTOPFN [Window Property]

If non-NIL, whenever the window is brought to the top, the TOTOPFN is called (with the window as a single argument). This function may be used to bring a collection of windows to the top together.

If the NOCALLTOPWFN argument of TOTOPW is non-NIL, the TOTOPFN of the window is not called, which provides a way of avoiding infinite loops when using TOTOPW from within a TOTOPFN.

Shrinking Windows Into Icons

Occasionally, a user will have a number of large windows on the screen, making it difficult to access those windows being used. To help with the problem of screen space management, the Interlisp-D window system allows the creation of Icons. An icon is a small rectangle (containing text or a bitmap) which is a "shrunken-down" form of a particular window. Using the Shrink and Expand window menu commands (see the beginning of this chapter), the user can shrink windows not currently being used into icons, and quickly restore the original windows at any time. This facility is controlled by the following functions and window properties:

(**SHRINKW** *WINDOW TOWHAT ICONPOSITION EXPANDFN*) [Function]

SHRINKW makes a small icon which represents *WINDOW* and removes *WINDOW* from the screen. Icons have a different window command menu that contains "EXPAND" instead of "SHRINK". The EXPAND command calls EXPANDW which returns the shrunken window to its original size and place. The icon can also be moved by pressing the LEFT button in it, or expanded by pressing the MIDDLE button in it.

The SHRINKFN property of the window *WINDOW* affects the operation of SHRINKW. If the SHRINKFN property of *WINDOW* is the atom DON'T, SHRINKW returns. Otherwise, the SHRINKFN property of the window is treated as a (list of) function(s) to apply to *WINDOW*; if any returns the atom DON'T, SHRINKW returns.

TOWHAT, if given, indicates the image the icon window will have. If *TOWHAT* is a string, atom or list, the icon's image will be that string (currently implemented as a title-only window with *TOWHAT* as the title.) If *TOWHAT* is a BITMAP, the icon's image will be a copy of the bitmap. If *TOWHAT* is a *WINDOW*, that window will be used as the icon.

If *TOWHAT* is not given (as is the case when invoked from the SHRINK window command), then the following apply in turn:

1. If the window has an ICONFN property, it gets called with the two arguments *WINDOW* and OLDICON, where *WINDOW* is the window being shrunk and OLDICON is the previously created icon, if any. The ICONFN should return one of the *TOWHAT* entities described above or return the OLDICON if it does not want to change it.

2. If the window has an ICON property, it is used as the value of *TOWHAT*.

3. If the window has neither an ICONFN or ICON property, the icon will be *WINDOW*'s title or, if *WINDOW* doesn't have a title, the date and time of the icon creation.

ICONPOSITION gives the position that the new icon will be on the screen. If it is NIL, the icon will be in the corner of the window furthest from the center of the screen.

In all but the default case, the icon is cached on the property ICONWINDOW of *WINDOW* so repeating SHRINKW reuses the same icon (unless overridden by the ICONFN described above). Thus to change the icon it is necessary to remove the ICONWINDOW property or call SHRINKW explicitly giving a *TOWHAT* argument.

(**EXPANDW** *ICONW*) [Function]

Restores the window for which *ICONW* is an icon, and removes the icon from the screen. If the EXPANDFN window property of the main window is the atom DON'T, the window won't be expanded. Otherwise, the window will be restored to its original size and location and the EXPANDFN (or list of functions) will be applied to it.

SHRINKFN [Window Property]

The SHRINKFN window property can be a single function or a list of functions that are called just before a window is shrunken by SHRINKW, with the window as a single argument. If any of the SHRINKFNs are the atom DON'T, or if the value returned by any of the SHRINKFNs is the atom DON'T, the window will not be shrunk.

EXPANDREGIONFN [Window property]

EXPANDREGIONFN, if non-NIL, should be the function to be called (with the window as its argument) before the window is actually expanded.

The EXPANDREGIONFN must return NIL or a valid region, and must not do any window operations (e.g., redisplaying). If NIL is returned, the window is expanded normally, as if the EXPANDREGIONFN had not existed. The region returned specifies the new region for the main window only, not for the group including any of its attached windows. The window will be opened in its new shape, and any attached windows will be repositioned or rejustified appropriately. The main window must have a REPAINTFN which can repaint the entire window under these conditions.

As with expanding windows normally, the OPENFN for the main window is not called.

Also, the window is reshaped without checking for a special shape function (e.g., a DOSHAPEFN).

ICONFN [Window Property]

If SHRINKW is called without begin given a TOWHAT argument (as is the case when invoked from the SHRINK window command) and the window's ICONFN property is non-NIL, then it gets called with two arguments, the window being shrunk and the previously created icon, if any. The ICONFN should return one of the TOWHAT entities described above or return the previously created icon if it does not want to change it.

ICON [Window Property]

If SHRINKW is called without being given a TOWHAT argument, the window's ICONFN property is NIL, and the ICON property is non-NIL, then it is used as the value of TOWHAT.

ICONWINDOW [Window Property]

Whenever an icon is created, it is cached on the property ICONWINDOW of the window, so calling SHRINKW again will reuse the same icon (unless overridden by the ICONFN.

Thus, to change the icon it is necessary to remove the ICONWINDOW property or call SHRINKW explicitly giving a TOWHAT argument.

DEFAULTICONFN [Variable]

Changes how an icon is created when a window having no ICONFN is shrunk or when SHRINKW, with a TOWHAT argument of a string, is called. The value of DEFAULTICONFN is a function of two arguments (window text); text is either NIL or a string. DEFAULTICONFN returns an icon window.

The initial value of DEFAULTICONFN is MAKETITLEBARICON. It creates a window that is a title bar only; the title is either the text argument, the window's title, or "Icon made <date>" for titleless windows. MAKETITLEBARICON places the title bar at some corner of the main window.

An alternative behavior is available by setting DEFAULTICONFN to be TEXTICON. TEXTICON creates a titled icon window from the text or window's title.

You can now copy-select titled icons such as those used by FileBrowser, SEdit, TEdit, Sketch. The default behavior is that the icon's title is unread (via BKSYSBUF), but if the icon window has a COPYFN property, that gets called instead, with the icon window as its argument. For example, if the name displayed in an icon is really a symbol, and you want copy selection to cause the name to be unread correctly with respect to the package and read table of the exec you are copying into, you could put the following COPYFN property on the icon window:

```
(LAMBDA (WINDOW)
```

```
(IL:BKSYSBUF  <fetch   symbolic  name  from  window>  T ))
```

EXPANDFN [Window Property]

The EXPANDFN window property can be a single function or a list of functions. If one of the EXPANDFNs is the atom DON'T, the window will not be expanded. Otherwise, the EXPANDFNs are called after the window has been expanded by EXPANDW, with the window as a single argument.

Creating Icons with ICONW

ICONW is a group of functions available for building small windows of arbitrary shape. These windows are principally for use as icons for shrinking windows; i.e., these functions are likely to be invoked from within the ICONFN of a window. An icon is specified by supplying its image (a bitmap) and a mask that specifies its shape. The mask is a bitmap of the same dimensions as the image whose bits are on (black) in those positions considered to be in the image, and off (white) in those positions where the background should show through. By using the mask and appropriate window functions, ICONW maintains the illusion that the icon window is nonrectangular, even though the actual window itself is rectangular. The illusion is not complete, of course. For example, if you try to select what looks like the background (or an occluded window) around the icon but still within its rectangular perimeter, the icon window itself is selected. Also, if you move a window occluded by an icon, the icon never notices that the background changed behind it. Icons created with ICONW can also have titles; some part of the image can be filled with text computed at the time the icon is created, or text may be changed after creation.

Creating Icons

Two types of icons can be created with ICONW, a borderless window containing an image defined by a mask and a window with a title.

(**ICONW** *IMAGE MASK POSITION NOOPENFLG*) [Function]

Creates a window at *POSITION*, or prompts for a position if *POSITION* is NIL. The window is borderless, and filled with *IMAGE*, as cookie-cut by *MASK*. If *MASK* is NIL, the image is considered rectangular (i.e., *MASK* defaults to a black bitmap of the same dimensions as *IMAGE*). If *NOOPENFLG* is T, the window is returned unopened.

(**TITLEDICONW** *ICON TITLE FONT POSITION NOOPENFLG JUST BREAKCHARS OPERATION*)
[Function]

Creates a titled icon at *POSITION*, or prompts for a position if *POSITION* is NIL. If *NOOPENFLG* is T, the window is returned unopened. The argument *ICON* is an instance of the record TITLEDICON, which specifies the icon image and mask, as with ICONW, and a region within the image to be used for displaying the title. Thus, the *ICON* argument is usually of the form

(create TITLEDICON ICON ← someIconImage

MASK ← iconMask TITLEREG ← someRegionWithinICON)

The title region is specified in coordinates relative to the icon, i.e., the lower-left corner of the image bitmap is (0, 0). The mask can be NIL if the icon is rectangular. The image should be white where it is covered by the title region. TITLEDICONW clears the region before printing on it. The title is printed into the specified region in the image, using *FONT*. If *FONT* is NIL it defaults to the value of DEFAULTICONFONT, initially Helvetica 10. The title is broken into multiple lines if necessary; TITLEDICONW attempts to place the breaks at characters that are in the list of character codes *BREAKCHARS*. *BREAKCHARS* defaults to (CHARCODE (SPACE ÿ)). In addition, line breaks are forced by any carriage returns in *TITLE*, independent of *BREAKCHARS*. *BREAKCHARS* is ignored if a long title would not otherwise fit in the specified region. For convenience, *BREAKCHARS* = FILE means the title is a file name, so break at file name field delimiters. The argument *JUST* indicates how the text should be justified relative to the region. It is an atom or list of atoms chosen from *TOP*, *BOTTOM*, *LEFT*, or *RIGHT*, which indicate the vertical positioning (flush to top or bottom) and/or horizontal positioning (flush to left edge or right). If *JUST* = NIL, the text is centered. The argument *OPERATION* is a display stream operation indicating how the title should be printed. If *OPERATION* is INVERT, then the title is printed white-on-black. The default *OPERATION* is REPLACE, meaning black-on-white. ERASE is the same as INVERT; PAINT is the same as REPLACE.

For convenience, TITLEDICONW can also be used to create icons that consist solely of a title, with no special image. If the argument *ICON* is NIL, TITLEDICONW creates a rectangular icon large enough to contain *TITLE*, with a border the same width as that on a regular window. The remaining arguments are as described above, except that a *JUST* of *TOP* or *BOTTOM* is not meaningful.

In the Medley release, TITLEDICONW can create icons with white text on a black background. To get this effect, your icon image must be black in the correct area, and you must specify the *OPERATION* argument as INVERT.

In Medley, you can copy- select the title of an icon.

Modifying Icons

(**ICONW.TITLE** *ICON TITLE*) [Function]

Returns the current title of the window *ICON*, which must be a window returned by TITLEDICONW. In addition, if *TITLE* is non-NIL, makes *TITLE* the new title of the window and repaints it accordingly. To erase the current title, make *TITLE* a null string.

(**ICONW.SHADE** *WINDOW SHADE*) [Function]

Returns the current shading of the window ICON, which must be a window returned by ICONW or TITLEDICONW. In addition, if *SHADE* is non-NIL, paints the texture *SHADE* on *WINDOW*. A typical use for this function is to communicate a change of state in a window that is shrunken, without reopening the window. To remove any shading, make *SHADE* be WHITESHADE.

Default Icons

When you shrink a window that has no ICONFN, the system currently creates an icon that looks like the window's title bar. You can make the system instead create titled icons by setting the global variable DEFAULTICONFN to the value TEXTICON.

(**TEXTICON** *WINDOW TEXT*) [Function]

Creates a titled icon window for the main window *WINDOW* containing the text *TEXT*, or the window's title if *TEXT* is NIL.

DEFAULTTEXTICON [Variable]

The value that TEXTICON passes to TITLEDICONW as its ICON argument. Initially it is NIL, which creates an unadorned rectangular window. However, you can set it to a TITLEDICON record of your choosing if you would like default icons to have a different appearance.

Coordinate Systems, Extents, And Scrolling

Note: The word "scrolling" has two distinct meanings when applied to Interlisp-D windows. This section documents the use of "scroll bars" on the left and bottom of a window to move an object displayed in the window. "Scrolling" also describes the feature where trying to print text off the bottom of a window will cause the contents to "scroll up." This second feature is controlled by the function DSPSCROLL (see Chapter 27).

One way of thinking of a window is as a "view" onto an object (e.g. a graph, a file, a picture, etc.) The object has its own natural coordinate system in terms of which its subparts are laid out. When the window is created, the X Offset and Y Offset of the window's display stream are set to map the origin of the object's coordinate system into the lower left point of the window's interior region. At the same time, the Clipping Region of the display stream is set to correspond to the interior of the window. From then on, the display stream's coordinate system is translated and its clipping region adjusted whenever the window is moved, scrolled or reshaped.

There are several distinct regions associated with a window viewing an object. First, there is a region in the window's coordinate system that contains the complete image of the object. This region (which can only be determined by application programs with knowledge of the "semantics" of the object) is stored as the EXTENT property of the window (below). Second, the clipping region of the display stream (obtainable with the function DSPCLIPPINGREGION, see Chapter 27) specifies the portion of the object that is actually visible in the window. This is set so that it corresponds to the interior of the window (not including the border or title). Finally, there is the region on the screen that specifies the total area that the window occupies, including the border and title. This region (in screen coordinates) is stored as the REGION property of the window (see the Miscellaneous Window Properties section below).

The window system supports the idea of scrolling the contents of a window. Scrolling regions are on the left and the bottom edge of each window. The LEFT button is used to indicate upward or leftward scrolling by the amount necessary to move the selected position to the top or the left edge. The RIGHT button is used to indicate downward or rightward scrolling by the amount necessary to move the top or left edge to the selected position. The MIDDLE button is used to indicate global placement of the object within the window (similar to "thumbing" a book). In the scroll region, the part of the object that is being viewed by the window is marked with a gray shade. If the whole scroll bar is thought of as the entire object, the shaded portion is the portion currently being viewed. This will only occur when the window "knows" how big the object is (see window property EXTENT, below).

When the button is released in a scroll region, the function SCROLLW is called. SCROLLW calls the scrolling function associated with the window to do the actual scrolling and provides a programmable entry to the scrolling operation.

(**SCROLLW** *WINDOW DELTAX DELTAY CONTINUOUSFLG*) [Function]

Calls the SCROLLFN window property of the window *WINDOW* with arguments *WINDOW*, *DELTAX*, *DELTAY* and *CONTINUOUSFLG*. See SCROLLFN window property below.

(**SCROLL.HANDLER** *WINDOW*) [Function]

This is the function that tracks the mouse while it is in the scroll region. It is called when the cursor leaves a window in either the left or downward direction. If N *MWINDOW* does not have a scroll region for this direction (e.g. the window has moved or reshaped since it was last scrolled), a scroll region is created that is SCROLLBARWIDTH wide. It then waits for SCROLLWAITTIME milliseconds and if the cursor is still inside the scroll region, it opens a window the size of the scroll region and changes the cursor to indicate the scrolling is taking place.

When a button is pressed, the cursor shape is changed to indicate the type of scrolling (up, down, left, right or thumb). After the button is held for WAITBEFORESCROLLTIME milliseconds, until the button is released SCROLLW is called each WAITBETWEENSCROLLTIME milliseconds. These calls are made with the CONTINUOUSFLG argument set to T. If the button is released before WAITBEFORESCROLLTIME milliseconds, SCROLLW is called with the CONTINUOUSFLG argument set to NIL.

The arguments passed to SCROLLW depend on the mouse button. If the LEFT button is used in the vertical scroll region, DY is distance from cursor position at the time the button was released to the top of the window and DX is 0. If the RIGHT button is used, the inverse of this quantity is used for DY and 0 for DX. If the LEFT button is used in the horizontal scroll region, DX is distance from cursor position to left of the window and DY is 0. If the RIGHT button is used, the inverse of this quantity is used for DX and 0 for DY.

If the MIDDLE button is pressed, the distance argument to SCROLLW will be a FLOATP between 0.0 and 1.0 that indicates the proportion of the distance the cursor was from the left or top edge to the right or bottom edge.

> Note: The scrolling regions will not come up if the window has a SCROLLFN window property of NIL, has a non-NIL NOSCROLLBARS window property, or if its SCROLLEXTENTUSE property has certain values and its EXTENT is fully visible.

(**SCROLLBYREPAINTFN** *WINDOW DELTAX DELTAY CONTINUOUSFLG*) [Function]

SCROLLBYREPAINTFN is the standard scrolling function which should be used as the SCROLLFN property for most scrolling windows.

This function, when used as a SCROLLFN, BITBLTs the bits that will remain visible after the scroll to their new location, fills the newly exposed area with texture, adjusts the window's coordinates and then calls the window's REPAINTFN on the newly exposed region. Thus this function will scroll any window that has a repaint function.

If *WINDOW* has an EXTENT property, SCROLLBYREPAINTFN will limit scrolling in the X and Y directions according to the value of the window property SCROLLEXTENTUSE.

If *DELTAX* or *DELTAY* is a FLOATP, SCROLLBYREPAINTFN will position the window so that its top or left edge will be positioned at that proportion of its EXTENT. If the window does not have an EXTENT, SCROLLBYREPAINTFN will do nothing.

If *CONTINUOUSFLG* is non-NIL, this indicates that the scrolling button is being held down. In this case, SCROLLBYREPAINTFN will scroll the distance of one linefeed height (as returned by DSPLINEFEED, see Chapter 27).

Scrolling is controlled by the following window properties:

EXTENT [Window Property]

Used to limit scrolling operations. Accesses the extent region of the window. If non-NIL, the EXTENT is a region in the window's display stream that contains the complete image of the object being viewed by the window. User programs are responsible for updating the EXTENT. The functions UNIONREGIONS, EXTENDREGION, etc. (see Chapter 27) are useful for computing a new extent region.

In some situations, it is useful to define an EXTENT that only exists in one dimension. This may be done by specifying an EXTENT region with a width or height of -1.

SCROLLFN handling recognizes this situation as meaning that the negative EXTENT dimension is unknown.

SCROLLFN [Window Property]

If the SCROLLFN property is NIL, the window will not scroll. Otherwise, it should be a function of four arguments: (1) the window being scrolled, (2) the distance to scroll in the horizontal direction (positive to right, negative to left), (3) the distance to scroll in the vertical direction (positive up, negative down), and (4) a flag which is T if the scrolling button is being held down. For more information, see SCROLL.HANDLER. For most scrolling windows, the SCROLLFN function should be SCROLLBYREPAINTFN.

NOSCROLLBARS [Window Property]

If the NOSCROLLBARS property is non-NIL, scroll bars will not be brought up for this window. This disables mouse-driven scrolling of a window. This window can still be scrolled using SCROLLW.

SCROLLEXTENTUSE [Window Property]

SCROLLBYREPAINTFN uses the SCROLLEXTENTUSE window property to limit how far scrolling can go in the X and Y directions. The possible values for SCROLLEXTENTUSE and their interpretations are:

NIL — This will keep the extent region visible or near visible. It will not scroll the window so that the top of the extent is below the top of the window, the bottom of the extent is more than one point above the top of the window, the left of the extent is to the right of the window and the right of the extent is to the left of the window. The EXTENT can be scrolled to just above the window to provide a way of "hiding" the contents of a window. In this mode the extent is either in the window or just of the top of the window.

T — The extent is not used to control scrolling. The user can scroll the window to anywhere. Having the EXTENT window property does all thumb scrolling to be supported so that the user can get back to the EXTENT by thumb scrolling.

LIMIT — This will keep the extent region visible. The window is only allowed to view within the extent.

+ — This will keep the extent region visible or just off in the positive direction in either X or Y (i.e., the image will be either be visible or just off to the top and/or right.)

- — This will keep the extent region visible or just off in the negative direction in either X or Y (i.e., the image will be either be visible or just off to the left and/or bottom).

+-

-+ This will keep the extent region visible or just off in the window (i.e. the image will be either be visible or just off to the left, bottom, top or right).

(XBEHAVIOR . YBEHAVIOR) If the SCROLLEXTENTUSE is a list, the CAR is interpreted as the scrolling limit in the X behavior and the CDR as the scrolling limit in the Y behavior. XBEHAVIOR and YBEHAVIOR should each be one of the atoms (NIL T LIMIT + - +- -+). The interpretations of the atoms is the same as above except that NIL is equivalent to LIMIT.

Note: The NIL value of SCROLLEXTENTUSE is equivalent to (LIMIT . +).

Example: If the SCROLLEXTENTUSE window property of a window (with an extent defined) is (LIMIT . T), the window will scroll uncontrolled in the Y dimension but be limited to the extent region in the X dimension.

Mouse Activity in Windows

The following window properties allow the user to control the response to mouse activity in a window. The value of these properties, if non-NIL, should be a function that will be called (with the window as argument) when the specified event occurs.

These functions should be "self-contained", communicating with the outside world solely via their window argument, e.g., by setting window properties. In particular, these functions should not expect to access variables bound on the stack, as the stack context is formally undefined at the time these functions are called. Since the functions are invoked asynchronously, they perform any terminal input/output operations from their own window.

WINDOWENTRYFN [Window Property]

Whenever a button goes down in the window and the process associated with the window is not the tty process, the WINDOWENTRYFN is called. The default is GIVE.TTY.PROCESS which gives the process associated with the window the tty and calls the BUTTONEVENTFN. WINDOWENTRYFN can be a list of functions and all will be called.

CURSORINFN [Window Property]

Whenever the mouse moves into the window, the CURSORINFN is called. If CURSORINFN is a list of functions, all will be called.

CURSOROUTFN [Window Property]

The CURSOROUTFN is called when the cursor leaves the window. If CURSOROUTFN is a list of functions, all will be called.

CURSORMOVEDFN [Window Property]

The CURSORMOVEDFN is called whenever the cursor has moved and is inside the window. CURSORMOVEDFN can be a list of functions and all will be called. This allows a window function to implement "active" regions within itself by having its CURSORMOVEDFN determine if the cursor is in a region of interest, and if so, perform some action.

BUTTONEVENTFN [Window Property]

The BUTTONEVENTFN is called whenever there is a change in the state (up or down) of the mouse buttons inside the window. Changes to the mouse state while the BUTTONEVENTFN is running will not be interpreted as new button events, and the BUTTONEVENTFN will not be re-invoked.

RIGHTBUTTONFN [Window Property]

The RIGHTBUTTONFN is called in lieu of the standard window menu operation (DOWINDOWCOM) when the RIGHT button is depressed in a window. More specifically, the RIGHTBUTTONFN is called instead of the BUTTONEVENTFN when (MOUSESTATE (ONLY RIGHT)). If the RIGHT button is to be treated like any other key in a window, supply RIGHTBUTTONFN and BUTTONEVENTFN with the same function.

When an application program defines its own RIGHTBUTTONFN, there is a convention that the default RIGHTBUTTONFN, DOWINDOWCOM , may be executed by pressing the RIGHT button when the cursor is in the header or border of a window. User RIGHTBUTTONFNs are encouraged to follow this convention, by calling DOWINDOWCOM if the cursor is not in the interior region of the window.

BACKGROUNDBUTTONEVENTFN [Variable]
BACKGROUNDCURSORINFN [Variable]
BACKGROUNDCURSOROUTFN [Variable]
BACKGROUNDCURSORMOVEDFN [Variable]

These variables provide a way of taking action when there is cursor action and the cursor in in the background. They are interpreted like the corresponding window properties. If set to the name of a function, that function will be called, respectively, whenever the cursor is in the background and a button changes, when the cursor moves into the background from a window, when the cursor moved from the background into a window and when the cursor moves from one place in the background to another.

Terminal I/O and Page Holding

Each process has its own terminal i/o stream (accessed as the stream T, see Chapter 25). The terminal i/o stream for the current process can be changed to point to a window by using the function TTYDISPLAYSTREAM, so that output and echoing of type-in is directed to a window.

(**TTYDISPLAYSTREAM** *DISPLAYSTREAM*) [Function]

Selects the display stream or window *DISPLAYSTREAM* to be the terminal output channel, and returns the previous terminal output display stream. TTYDISPLAYSTREAM puts

DISPLAYSTREAM into scrolling mode and calls PAGEHEIGHT with the number of lines that will fit into *DISPLAYSTREAM* given its current Font and Clipping Region. The line length of TTYDISPLAYSTREAM is computed (like any other display stream) from its Left Margin, Right Margin, and Font. If one of these fields is changed, its line length is recalculated. If one of the fields used to compute the number of lines (such as the Clipping Region or Font) changes, PAGEHEIGHT is not automatically recomputed. (TTYDISPLAYSTREAM (TTYDISPLAYSTREAM)) will cause it to be recomputed.

If the window system is active, the line buffer is saved in the old TTY window, and the line buffer is set to the one saved in the window of the new display stream, or to a newly created line buffer (if it does not have one). Caution: It is possible to move the TTYDISPLAYSTREAM to a nonvisible display stream or to a window whose current position is not in its clipping region.

(**PAGEHEIGHT** *N*) [Function]

If *N* is greater than 0, it is the number of lines of output that will be printed to TTYDISPLAYSTREAM before the page is held. A page is held before the *N*+1 line is printed to TTYDISPLAYSTREAM without intervening input if there is no terminal input waiting to be read. The output is held with the screen video reversed until a character is typed. Output holding is disabled if *N* is 0. PAGEHEIGHT returns the previous setting.

PAGEFULLFN [Window Property]

If the PAGEFULLFN window property is non-NIL, it will be called with the window as a single argument when the window is full (i.e., when enough has been printed since the last TTY interaction so that the next character printed will cause information to be scrolled off the top of the window.)

If the PAGEFULLFN window property is NIL, the system function PAGEFULLFN is called. PAGEFULLFN simply returns if there are characters in the type-in buffer for WINDOW, otherwise it inverts the window and waits for the user to type a character. PAGEFULLFN is user advisable.

Note: The PAGEFULLFN window property is only called on windows which are the TTYDISPLAYSTREAM of some process.

TTY Process and the Caret

At any time, one process is designated as the TTY process, which is used for accepting keyboard input. The TTY process can be changed to a given process by calling GIVE.TTY.PROCESS (see Chapter 23), or by clicking the mouse in a window associated with the process. The latter mechanism is implemented with the following window property:

PROCESS [Window Property]

If the PROCESS window property is non-NIL, it should be a PROCESS and will be made the TTY process by GIVE.TTY.PROCESS (see Chapter 23), the default

WINDOWENTRYFN property (see above). This implements the mechanism by which the keyboard is associated with different processes.

The window system uses a flashing caret (^) to indicate the position of the next window typeout. There is only one caret visible at any one time. The caret in the current TTY process is always visible; if it is hidden by another window, its window is brought to the top. An exception to this rule is that the flashing caret's window is not brought to the top if the user is buttoning or has a shift key down. This prevents the destination window (which has the tty and caret flashing) from interfering with the window one is trying to select text to copy from.

(**CARET** *NEWCARET*) [Function]

Sets the shape that blinks at the location of the next output to the current process. *NEWCARET* should be one of the following:

a CURSOR object	If *NEWCARET* is a CURSOR object (see Chapter 30), it is used to give the new caret shape
OFF	Turns the caret off
NIL	The caret is not changed. CARET returns a CURSOR representing the current caret
T	Reset the caret to the value of DEFAULTCARET. DEFAULTCARET can be set to change the initial caret for new processes.

The hotspot of *NEWCARET* indicates which point in the new caret bitmap should be located at the current output position. The previous caret is returned. Note: the bitmap for the caret is not limited to the dimensions CURSORWIDTH by CURSORHEIGHT.

(**CARETRATE** *ONRATE OFFRATE*) [Function]

Sets the rate at which the caret for the current process will flash. The caret will be visible for *ONRATE* milliseconds, then not visible for *OFFRATE* milliseconds. If *OFFRATE* is NIL then it is set to be the same as *ONRATE*. If *ONRATE* is T, both the "on" and "off" times are set to the value of the variable DEFAULTCARETRATE (initially 333). The previous value of CARETRATE is returned. If the caret is off, CARETRATE return NIL.

Miscellaneous Window Functions

(**CLEARW** *WINDOW*) [Function]

Fills *WINDOW* with its background texture, changes its coordinate system so that the origin is the lower left corner of the window, sets its X position to the left margin and sets its Y position to the base line of the uppermost line of text, ie. the top of the window less the font ascent.

(**INVERTW** *WINDOW SHADE*) [Function]

Fills the window *WINDOW* with the texture *SHADE* in INVERT mode. If *SHADE* is NIL, BLACKSHADE is used. INVERTW returns *WINDOW* so that it can be used inside RESETFORM.

(**FLASHWINDOW** *WIN? N FLASHINTERVAL SHADE*) [Function]

Flashes the window *WIN?* by "inverting" it twice. N is the number of times to flash the window (default is 1). *FLASHINTERVAL* is the length of time in milliseconds to wait between flashes (default is 200). *SHADE* is the shade that will be used to invert the window (default is BLACKSHADE).

If *WIN?* is NIL, the whole screen is flashed. In this case, the *SHADE* argument is ignored (can only invert the screen).

(**WHICHW** *X Y*) [Function]

Returns the window which contains the position in screen coordinates of *X* if *X* is a POSITION , the position (*X, Y*) if *X* and *Y* are numbers, or the position of the cursor if *X* is NIL. Returns NIL if the coordinates are not in any window. If they are in more than one window, it returns the uppermost.

Example: (WHICHW) returns the window that the cursor is in.

(**DECODE/WINDOW/OR/DISPLAYSTREAM** *DSORW WINDOWVAR TITLE BORDER*) [Function]

Returns a display stream as determined by the *DSORW* and *WINDOWVAR* arguments. If *DSORW* is a display stream, it is returned. If *DSORW* is a window, its display stream is returned. If *DSORW* is NIL, the litatom *WINDOWVAR* is evaluated. If its value is a window, its display stream is returned. If its value is not a window, *WINDOWVAR* is set to a newly created window (prompting user for region) whose display stream is then returned. If *DSORW* is NEW, the display stream of a newly created window is returned. If a *window* is involved in the decoding, it is opened and if *TITLE* or *BORDER* are given, the *TITLE* or *BORDER* property of the window are reset. The *DSORW* = NIL case is most useful for programs that want to display their output in a window, but want to reuse the same window each time they are called. The non-NIL cases are good for decoding a display stream argument passed to a function.

(**WIDTHIFWINDOW** *INTERIORWIDTH BORDER*) [Function]

Returns the width of the window necessary to have INTERIORWIDTH points in its interior if the width of the border is *BORDER*. If *BORDER* is NIL, the default border size WBorder is used.

(**HEIGHTIFWINDOW** *INTERIORHEIGHT TITLEFLG BORDER*) [Function]

Returns the height of the window necessary to have *INTERIORHEIGHT* points in its interior with a border of *BORDER* and, if *TITLEFLG* is non-NIL, a title. If *BORDER* is NIL, the default border size WBorder is used.

WIDTHIFWINDOW and HEIGHTIFWINDOW are useful for calculating the width and height for a call to GETBOXPOSITION for the purpose of positioning a prospective window.

(**MINIMUMWINDOWSIZE** *WINDOW*) [Function]

Returns a dotted pair, the CAR of which is the minimum width *WINDOW* needs and the CDR or which is the minimum height *WINDOW* needs.

The minimum size is determined by the value of the window property MINSIZE of *WINDOW*. If the value of the MINSIZE window property is NIL, the width is 26 and the height is the height *WINDOW* needs to have its title, border and one line of text visible. If MINSIZE is a dotted pair, it is returned. If it is a litatom, it should be a function which is called with *WINDOW* as its first argument, which should return a dotted pair.

Miscellaneous Window Properties

TITLE [Window Property]

Accesses the title of the window. If a title is added to a window whose title is NIL or the title is removed (set to NIL) from a window with a title, the window's exterior (its region on the screen) is enlarged or reduced to accomodate the change without changing the window's interior. For example, (WINDOWPROP WINDOW 'TITLE "Results") changes the title of WINDOW to be "Results". (WINDOWPROP WINDOW 'TITLE NIL) removes the title of WINDOW.

BORDER [Window Property]

Accesses the width of the border of the window. The border will have at most 2 point of white (but never more than half) and the rest black. The default border is the value of the global variable WBorder (initially 4).

WINDOWTITLESHADE [Window Property]

Accesses the window title shade of the window. If non-NIL, it should be a texture which is used as the "backgound texture" for the title bar on the top of the window. If it is NIL, the value of the global variable WINDOWTITLESHADE (initially BLACKSHADE) is used. Note that black is always used as the background of the title printed in the title bar, so that the letters can be read. The remaining space is painted with the "title shade".

HARDCOPYFN [Window Property]

If non-NIL, it should be a function that is called by the window menu command Hardcopy to print the contents of a window. The HARDCOPYFN property is called with two arguments, the window and an image stream to print to. If the window does not have a HARDCOPYFN, the bitmap image of the window (including the border and title) are printed on the file or printer.

DSP [Window Property]

Value is the display stream of the window. All system functions will operate on either the window or its display stream. This window property cannot be changed using WINDOWPROP.

HEIGHT [Window Property]
WIDTH [Window Property]

Value is the height and width of the interior of the window (the usable space not counting the border and title). These window properties cannot be changed using WINDOWPROP.

REGION [Window Property]

Value is a region (in screen coordinates) indicating where the window (counting the border and title) is located on the screen. This window property cannot be changed using WINDOWPROP.

Example: A Scrollable Window

The following is a simple example showing how one might create a scrollable window.

CREATE.PPWINDOW creates a window that displays the pretty printed expression EXPR. The window properties PPEXPR, PPORIGX, and PPORIGY are used for saving this expression, and the initial window position. Using this information, REPAINT.PPWINDOW simply reinitializes the window position, and prettyprints the expression again. Note that the whole expression is reformatted every time, even if only a small part actually lies within the window. If this window was going to be used to display very large structures, it would be desirable to implement a more sophisticated REPAINTFN that only redisplays that part of the expression within the window. However, this scheme would be satisfactory if most of the items to be displayed are small.

RESHAPE.PPWINDOW resets the window (and stores the initial window position), calls REPAINT.PPWINDOW to display the window's expression, and then sets the EXTENT property of the window so that SCROLLBYREPAINTFN will be able to handle scrolling and "thumbing" correctly.

```
(DEFINEQ

(CREATE.PPWINDOW
  [LAMBDA (EXPR)              (* rrb " 4-OCT-82 12:06")
                         (* creates a window that displays
                            a pretty printed expression.)

(PROG (WINDOW)
                         (* ask the user for a piece of the
                            screen and make it into a window.)
  (SETQ WINDOW (CREATEW NIL "PP window"))
                         (* put the expression on the
                            property list of the window so that
                            the repaint and reshape functions
                            can access it.)
  (WINDOWPROP WINDOW (QUOTE PPEXPR) EXPR)
                         (* set the repaint and reshape
                            functions.)
```

```
    (WINDOWPROP WINDOW (QUOTE REPAINTFN)
        (FUNCTION REPAINT.PPWINDOW))
    (WINDOWPROP WINDOW (QUOTE RESHAPEFN)
        (FUNCTION RESHAPE.PPWINDOW))
                    (* make the scroll function
                       SCROLLBYREPAINTFN, a system
                       function that uses the repaint
                       function to do scrolling.)
    (WINDOWPROP WINDOW (QUOTE SCROLLFN)
         (FUNCTION SCROLLBYREPAINTFN))
                    (* call the reshape function to
                       initially print the expression and
                       calculate its extent.)
     (RESHAPE.PPWINDOW WINDOW)
     (RETURN WINDOW])


(REPAINT.PPWINDOW
 [LAMBDA (WINDOW REGION)          (* rrb " 4-OCT-82 11:52")

        (* the repainting function for a window with a
           pretty printed expression.  This repainting
           function ignores the region to be repainted
           and repaints the entire window.)

                    (* set the window position to the
                       beginning of the pretty printing
                       of the expression.)
      (MOVETO (WINDOWPROP WINDOW (QUOTE PPORIGX))
              (WINDOWPROP WINDOW (QUOTE PPORIGY))
              WINDOW)
      (PRINTDEF (WINDOWPROP WINDOW (QUOTE PPEXPR))
                0 NIL NIL NIL WINDOW])


(RESHAPE.PPWINDOW
  [LAMBDA (WINDOW)               (* rrb " 4-OCT-82 12:01")
                    (* the reshape function for a
                       window with a pretty printed
                       expression.)
      (PROG (BTM)

        (* set the position of the window so that the
           first character appears in the upper left corner
           and save the X and Y for the repaint function.)

    (DSPRESET WINDOW)
    (WINDOWPROP WINDOW (QUOTE PPORIGX)
         (DSPXPOSITION NIL WINDOW))
    (WINDOWPROP WINDOW (QUOTE PPORIGY)
         (DSPYPOSITION NIL WINDOW))
                    (* call the repaint function to
                       pretty print the expression in
```

```
                   the newly cleared window.)
   (REPAINT.PPWINDOW WINDOW)

      (* save the region actually covered by the pretty
       printed expression so that the scrolling routines
       will know where to stop.  The pretty printing of
       the expression does a carriage return after the
       last piece of the expression printed so that the
       current position is the base line of the next line
       of text.  Hence the last visible piece of the
       expression (BTM) is the ending position plus the
       height of the font above the base line (its ASCENT).)

  (WINDOWPROP WINDOW (QUOTE EXTENT)
     create REGION
           LEFT ← 0
           BOTTOM ← [SETQ BTM (IPLUS
                   (DSPYPOSITION NIL WINDOW)
                   (FONTPROP WINDOW (QUOTE ASCENT]
           WIDTH ←(WINDOWPROP WINDOW (QUOTE WIDTH))
           HEIGHT ←(IDIFFERENCE
                   (WINDOWPROP WINDOW (QUOTE HEIGHT))
                   BTM])
)
```

Menus

A menu is basically a means of selecting from a list of items. The system provides common layout and interactive user selection mechanisms, then calls a user-supplied function when a selection has been confirmed. The two major constituents of a menu are a list of items and a "when selected function." The label that appears for each item is the item itself for non-lists, or its CAR if the item is a list. In addition, there are a multitude of different formatting parameters for specifying font, size, and layout. When a menu is created, its unspecified fields are filled with defaults and its screen image is computed and saved.

Menus can be either pop up or fixed. If fixed menus are used, the menu must be included in a window.

(**MENU** *MENU POSITION RELEASECONTROLFLG —*) [Function]

This function provides menus that pop up when they are used. It displays *MENU* at *POSITION* (in screen coordinates) and waits for the user to select an item with a mouse key. Before any mouse key is pressed, the item the mouse is over is boxed. After any key is down, the selected menu item is video reversed. When all keys are released, *MENU*'s WHENSELECTEDFN field is called with four arguments: (1) the item selected, (2) the menu, (3) the last mouse key released (LEFT, MIDDLE, or RIGHT), and (4) the reverse list of superitems rolled through when selecting the item and *MENU* returns its value. If no item is selected, *MENU* returns NIL. If *POSITION* is NIL, the menu is brought up at the value from *MENU*'s MENUPOSITION field, if it is a *POSITION*, or at the current cursor position. The orientation of *MENU* with respect to the specified position is determined by its MENUOFFSET field.

If *RELEASECONTROLFLG* is NIL, this process will retain control of the mouse. In this case, if the user lets the mouse key up outside of the menu, *MENU* return NIL. (Note: this is the standard way of allowing the user to indicate that they do not want to make the offered choice.) If *RELEASECONTROLFLG* is non-NIL, this process will give up control of the mouse when it is outside of the menu so that other processes can be run. In this case, clicking outside the menu has no effect on the call to *MENU*. If the menu is closed (for example, by right buttoning in it and selecting "Close" from the window menu), *MENU* returns NIL. Programmers are encouraged to provide a menu item such as "cancel" or "abort" which gives users a positive way of indicating "no choice".

Note: A "released" menu will stay visible (on top of the window stack) until it is closed or an item is selected.

(**ADDMENU** *MENU WINDOW POSITION DONTOPENFLG*) [Function]

This function provides menus that remain active in windows. ADDMENU displays *MENU* at *POSITION* (in window coordinates) in *WINDOW*. If the window is too small to display the entire menu, the window is made scrollable. When an item is selected, the value of the WHENSELECTEDFN field of *MENU* is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse button that the item was selected with (LEFT, MIDDLE, or RIGHT). More than one menu can be put in a window, but a menu can only be added to one window at a time. ADDMENU returns the window into which *MENU* is placed.

If *WINDOW* is NIL, a window is created at the position specified by *POSITION* (in screen coordinates) that is the size of *MENU*. If a window is created, it will be opened unless *DONTOPENFLG* is non-NIL. If *POSITION* is NIL, the menu is brought up at the value of *MENU*'s MENUPOSITION field (in window coordinates), if it is a position, or else in the lower left corner of *WINDOW*. If both *WINDOW* and *POSITION* are NIL, a window is created at the current cursor position.

Warning: ADDMENU resets several of the window properties of *WINDOW*. The CURSORINFN, CURSORMOVEDFN, and BUTTONEVENTFN window properties are replaced with MENUBUTTONFN, so that *MENU* will be active. MENUREPAINTFN is added to the REPAINTFN window property to update the menu image if the window is redisplayed. The SCROLLFN window property is changed to SCROLLBYREPAINTFN if the window is too small for the menu, to make the window scroll.

(**DELETEMENU** *MENU CLOSEFLG FROMWINDOW*) [Function]

This function removes *MENU* from the window *FROMWINDOW*. If *MENU* is the only menu in the window and *CLOSEFLG* is non-NIL, its window will be closed (by CLOSEW).

If *FROMWINDOW* is NIL, the list of currently open windows is searched for one that contains *MENU*. If none is found, DELETEMENU does nothing.

Menu Fields

A menu is a datatype with the following fields:

ITEMS [Menu Field]

The list of items to appear in the menu. If an item is a list, its CAR will appear in the menu. If the item (or its CAR) is a bitmap, the bitmap will be displayed in the menu. The default selection functions interpret each item as a list of three elements: a label, a form whose value is returned upon selection, and a help string that is printed in the prompt window when the user presses a mouse key with the cursor pointing to this item. The default subitem function interprets the fourth element of the list. If it is a list whose CAR is the litatom SUBITEMS , the CDR is taken as a list of subitems.

SUBITEMFN [Menu Field]

A function to be called to determine if an item has any subitems. If an item has subitems and the user rolls the cursor out the right of that item, a submenu with that item's subitems in it pops up. If the user selects one of the items from the submenu, the selected subitem is handled as if it were selected from the main menu. If the user rolls out of the submenu to the left, the submenu is taken down and selection resumes from the main menu.

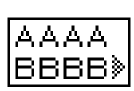
An item with subitems is marked in the menu by a grey, right pointing triangle following the label.

The function is called with two arguments: (1) the menu and (2) the item. It should return a list of the subitems of this item if any. (It is called twice to compute the menu image and each time the user rolls out of the item box so it should be moderately efficient. The default SUBITEMFN, DEFAULTSUBITEMFN, checks to see if the item is a list whose fourth element is a list whose CAR is the litatom SUBITEMS and if so, returns the CDR of it.

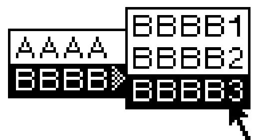
For example:

```
(create MENU
    ITEMS ← '(AAAA (BBBB 'BBBB "help string for
BBBB"
              (SUBITEMS BBBB1 BBBB2 BBBB3))))
```

will create a menu with items A and B in which B will have subitems B1, B2 and B3. The following picture below shows this menu as it first appears:



The following picture shows the submenu, with the item BBBB3 selected by the cursor ():



WHENSELECTEDFN [Menu Field]

A function to be called when an item is selected. The function is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (LEFT, MIDDLE, or RIGHT). The default function DEFAULTWHENSELECTEDFN evaluates and returns the value of the second element of the item if the item is a list of at least length 2. If the item is not a list of at least length 2, DEFAULTWHENSELECTEDFN returns the item.

Note: If the menu is added to a window with ADDMENU, the default WHENSELECTEDFN is BACKGROUNDWHENSELECTEDFN, which is the same as DEFAULTWHENSELECTEDFN except that EVAL.AS.PROCESS is used to evaluate the second element of the item, instead of tying up the mouse process.

WHENHELDFN [Menu Field]

The function which is called when the user has held a mouse key on an item for MENUHELDWAIT milliseconds (initially 1200). The function is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (LEFT, MIDDLE, or RIGHT). WHENHELDFN is intended for prompting users. The default is DEFAULTMENUHELDFN which prints (in the prompt window) the third element of the item or, if there is not a third element, the string "This item will be selected when the button is released."

WHENUNHELDFN [Menu Field]

If WHENHELDFN was called, WHENUNHELDFN will be called: (1) when the cursor leaves the item, (2) when a mouse key is released, or (3) when another key is pressed. The function is called with the same three argument values used to call WHENHELDFN. The default WHENUNHELDFN is the function CLRPROMPT, which just clears the prompt window.

MENUPOSITION [Menu Field]

The position of the menu to be used if the call to MENU or ADDMENU does not specify a position. For popup menus, this is in screen coordinates. For fixed menus, it is in the coordinates of the window the menu is in. The point within the menu image that is placed at this position is determined by MENUOFFSET. If MENUPOSITION is NIL, the menu will be brought up at the cursor position.

MENUOFFSET [Menu Field]

The position in the menu image that is to be located at MENUPOSITION. The default offset is (0,0). For example, to bring up a menu with the cursor over a particular menu item, set its MENUOFFSET to a position within that item and set its MENUPOSITION to NIL.

MENUFONT [Menu Field]

The font in which the items will be appear in the menu. Default is the value of MENUFONT.

TITLE [Menu Field]

If non-NIL, the value of this field will appear as a title in a line above the menu.

MENUTITLEFONT [Menu Field]

The font in which the title of the menu will be appear. If this is NIL, the title will be in the same font as window titles. If it is T, it will be in the same font as the menu items.

CENTERFLG [Menu Field]

If non-NIL, the menu items are centered; otherwise they are left-justified.

MENUROWS [Menu Field]
MENUCOLUMNS [Menu Field]

These fields control the shape of the menu in terms of rows and columns. If MENUROWS is given, the menu will have that number of rows. If MENUCOLUMNS is given, the menu will have that number of columns. If only one is given, the other one will be calculated to generate the minimal rectangular menu. (Normally only one of MENUROWS or MENUCOLUMNS is given.) If neither is given, the items will be in one column.

ITEMHEIGHT [Menu Field]

The height of each item box in the menu. If not specified, it will be the maximum of the height of the MENUFONT and the heights of any bitmaps appearing as labels.

ITEMWIDTH [Menu Field]

The width of each item box in the menu. If not specified, it will be the width of the largest item in the menu.

MENUBORDERSIZE [Menu Field]

The size of the border around each item box. If not specified, 0 (no border) is used.

MENUOUTLINESIZE [Menu Field]

The size of the outline around the entire menu. If not specified, a maximum of 1 and the MENUBORDERSIZE is used.

CHANGEOFFSETFLG [Menu Field]

(popup menus only) If CHANGEOFFSETFLG is non-NIL, the position of the menu offset is set each time a selection is confirmed so that the menu will come up next time in the same position relative to the cursor. This will cause the menu to reappear in the same place on the screen if the cursor has not moved since the last selection. This is implemented by changing the MENUOFFSET field on each use. If CHANGEOFFSETFLG is the atom X or the atom Y, only the X or the Y coordinate of the MENUOFFSET field will be changed. For example, by setting the MENUOFFSET position to (-1,0) and setting CHANGEOFFSETFLG to Y, the menu will pop up so that the cursor is just to the left of the last item selected. This is the setting of the window command menus.

The following fields are read only.

IMAGEHEIGHT [Menu Field]

Returns the height of the entire menu.

IMAGEWIDTH [Menu Field]

Returns the width of the entire menu.

Miscellaneous Menu Functions

(**MAXMENUITEMWIDTH** *MENU*) [Function]

Returns the width of the largest menu item label in the menu *MENU*.

(**MAXMENUITEMHEIGHT** *MENU*) [Function]

Returns the height of the largest menu item label in the menu *MENU*.

(**MENUREGION** *MENU*) [Function]

Returns the region covered by the image of *MENU* in its window.

(**WFROMMENU** *MENU*) [Function]

Returns the window *MENU* is located in, if it is in one; NIL otherwise.

(**DOSELECTEDITEM** *MENU ITEM BUTTON*) [Function]

Calls *MENU*'s WHENSELECTEDFN on *ITEM* and *BUTTON*. It provides a programmatic way of making a selection. It does not change the display.

(**MENUITEMREGION** *ITEM MENU*) [Function]

Returns the region occupied by *ITEM* in *MENU*.

(**SHADEITEM** *ITEM MENU SHADE DS/W*) [Function]

Shades the region occupied by *ITEM* in *MENU*. If *DS/W* is a display stream or a window, it is assumed to be where *MENU* is displayed. Otherwise, WFROMMENU is called to locate the window *MENU* is in. Shading is persistent, and is reapplied when the window the menu is in gets redisplayed. To unshade an item, call with a *SHADE* of 0.

(**PUTMENUPROP** *MENU PROPERTY VALUE*) [Function]

Stores the property *PROPERTY* with the value *VALUE* on a property list in the menu *MENU*. The user can use this property list for associating arbitrary data with a menu object.

(**GETMENUPROP** *MENU PROPERTY*) [Function]

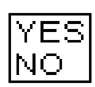
Returns the value of the *PROPERTY* property of the menu *MENU*.

Examples of Menu Use

Example: A simple menu:

```
(MENU (create MENU ITEMS _ '((YES T) (NO (QUOTE
NIL))) ))
```

Creates a menu with items YES and NO in a single vertical column:



If YES is selected, T will be returned. Otherwise, NIL will be returned.

Example: A simple menu, with centering:

```
(MENU (create MENU TITLE ← "Foo?"
     ITEMS ← '((YES T "Adds the Foo feature.")
               (NO 'NO "Removes the Foo feature."))
            CENTERFLG ← T))
```

Creates a menu with a title Foo? and items YES and NO centered in a single vertical column:

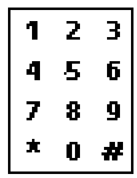
The strings following the YES and NO are help strings and will be printed if the cursor remains over one of the items for a period of time. This menu differs from the one above in that it distinquishes the NO case from the case where the user clicked outside of the menu. If the user clicks outside of the menu, NIL is returned.

Example: A multi-column menu:

```
(create MENU ITEMS ← '(1 2 3 4 5 6 7 8 9 * 0 #)
            CENTERFLG ← T
```

```
MENUCOLUMNS ← 3
MENUFONT ← (FONTCREATE 'MODERN 10 'BOLD)
ITEMHEIGHT ← 15
ITEMWIDTH ← 15
CHANGEOFFSETFLG ← T)
```

Creates a touch-tone-phone number pad with the items in 15 by 15 boxes printed in Modern 10 bold font:



If used in pop up mode, its first use will have the cursor in the middle. Subsequent use will have the cursor in the same relative location as the previous selection.

Example: A program using a previously-saved menu:

```
(SELECTQ [MENU
   (COND ((type? MENU FOOMENU)
         (* use previously computed menu.)
       FOOMENU)
        (T (* create and save the menu)
      (SETQ FOOMENU
        (create MENU
            ITEMS ← '((A 'A-SELECTED "prompt string
  for A")
                 (B 'B-SELECTED "prompt string for B"]
     (A-SELECTED (* if A is selected) (DOATHING))
     (B-SELECTED (* if B is selected) (DOBTHING))
      (PROGN (* user selected outside the menu) NIL)))
```

This expression displays a pop up menu with two items, A and B, and waits for the user to select one. If A is selected, DOATHING is called. If B is selected, DOBTHING is called. If neither of these is selected, the form returns NIL.

The purpose of this example is to show some good practices to follow when using menus. First, the menu is only created once, and saved in the variable FOOMENU. This is more efficient if the menu is used more than once. Second, all of the information about the menu is kept in one place, which makes it easy to understand and edit. Third, the forms evaluated as a result of selecting something from the menu are part of the code and hence will be known to masterscope (as opposed to the situation if the forms were stored as part of the items). Fourth, the items in the menu have help strings for the user. Finally, the code is commented (always worth the trouble).

Free Menus

Free Menus are powerful and flexible menus that are useful for applications needing menus with different types of items, including command items, state items, and items that can be edited. A Free Menu is part of a window. It can can be opened and closed as desired, or attached as a control menu to the application window.

Making a Free Menu

A Free Menu is built from a description of the contents and layout of the menu. As a Free Menu is simply a group of items, a Free Menu Description is simply a specification of a group of items. Each group has properties associated with it, as does each Free Menu Item. These properties specify the format of the items in the group, and the behavior of each item. The function `FREEMENU` takes a Free Menu Description, and returns a closed window with the Free Menu in it.

The easiest way to make a Free Menu is to define a specific function which calls `FREEMENU` with the Free Menu Description in the function. This function can then also set up the Free Menu window as required by the application. The Free Menu Description is saved as part of the specific function when the application is saved. Alternately, the Free Menu Description can be saved as a variable in your file; then just call `FREEMENU` with the name of the variable. This may be a more difficult alternative if the backquote facility is used to build the Free Menu Description.

Free Menu Formatting

A Free Menu can be formatted in one of four ways. The items in any group can be automatically laid out in rows, in columns, or in a table, or else the application can specify the exact location of each item in the group. Free Menu keeps track of the region that a group of items occupies, and items can be justified within that region. This way an item can be automatically positioned at one of the nine justification locations, top-left, top-center, top-right, middle-left, etc.

Free Menu Description

A Free Menu Description, specifying a group of items, is a list structure. The first entry in the list is an optional list of the properties for this group of items. This entry is in the form:

```
(PROPS   <PROP> <VALUE> <PROP> <VALUE> ...)
```

The keyword `PROPS` determines whether or not the optional group properties list is specified..

One important group property is `FORMAT`. The four types of formatting, `ROW`, `TABLE`, `COLUMN`, or `EXPLICIT`, determine the syntax of the rest of the Free Menu Description. When using `EXPLICIT` formatting, the rest of the description is any number of Item Descriptions which have `LEFT` and `BOTTOM` properties specifying the position of the item in the menu. The syntax is:

```
((PROPS FORMAT EXPLICIT ...)
 <ITEM DESCRIPTION>
 <ITEM DESCRIPTION> ...)
```

When using ROW or TABLE formatting, the rest of the description is any number of item groups, each group corresponding to a row in the menu. These groups are identical in syntax to an EXPLICIT group description. The groups have an optional PROPS list and any number of Item Descriptions. The items need not have LEFT and BOTTOM properties, as the location of each item is determined by the formatter. However, the order of the rows and items is important. The menu is laid out top to bottom by row, and left to right within each row. The syntax is:

```
((PROPS FORMAT ROW ...)      ; props of this group
 (<ITEM DESCRIPTION>         ; items in first row
  <ITEM DESCRIPTION> ...)
  ((PROPS ...)               ; props of second row
   <ITEM DESCRIPTION>        ; items in second row
   <ITEM DESCRIPTION> ...))
```

(The comments above only describe the syntax.)

For COLUMN formatting, the syntax is identical to that of ROW formatting. However, each group of items corresponds to a column in the menu, rather than a row. The menu is laid out left to right by column, top to bottom within each column.

Finally, a Free Menu Description can have recursively nested groups. Anywhere the description can take an Item Description, it can take a group, marked by the keyword GROUP. A nested group inherits all of the properties of its mother group, by default. However, any of these properties can be overridden in the nested groups PROPS list, including the FORMAT. The syntax is:

```
(                ; no PROPS list, default row format
(<ITEM DESCRIPTION>          ; first in row
(GROUP           ; nested group, second in row
   (PROPS FORMAT COLUMN ...) ; optional props
   (<ITEM DESCRIPTION> ...)  ; first column
   (<ITEM DESCRIPTION> ...))
    <ITEM DESCRIPTION>))     ; third in row
```

Here is an example of a simple Free Menu Description for a menu which might provide access to a simple data base:

```
(((LABEL LOOKUP SELECTEDFN MYLOOKUPFN)
  (LABEL EXIT SELECTEDFN MYEXITFN))
 ((LABEL Name: TYPE DISPLAY) (LABEL "" TYPE EDIT ID NAME))
 ((LABEL Address: TYPE DISPLAY) (LABEL "" TYPE EDIT ID ADDRESS))
 ((LABEL Phone: TYPE DISPLAY)
```

```
  (LABEL "" TYPE EDIT LIMITCHARS MYPHONEP ID PHONE)))
```

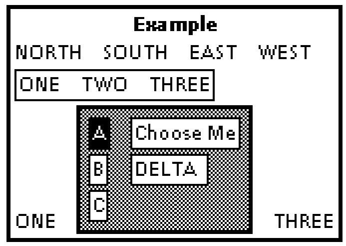
This menu has two command buttons, LOOKUP and EXIT, and three edit fields, with IDs NAME, PHONE, and ADDRESS. The Edit items are initialized to the empty string, as in this example they need no other initial value. The user could select the Name: prompt, type a person’s name, and then press the LOOKUP button. The function MYLOOKUPFN would be called. That function would look at the NAME Edit item, look up that name in the data base, and fill in the rest of the fields appropriately. The PHONE item has MYPHONEP as a LIMITCHARS function. This function would be called when editing the phone number, in order to restrict input to a valid phone number. After looking up Perry, the Free Menu might look like:

```
LOOKUP  EXIT
Name:  Herbert Q Perry
Address:  13 Middleperry Dr
Phone:  (411) 767-1234
```

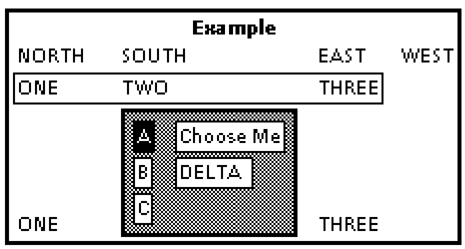
Here is a more complicated example:

```
((PROPS FONT (MODERN 10))
 ((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))
 ((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))
 ((PROPS ID ROW3 BOX 1)
  (LABEL ONE) (LABEL TWO) (LABEL THREE))
 ((PROPS ID ROW4)
  (LABEL ONE ID ALPHA)
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
     ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT
T))
     (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
     (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
       ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
           INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
     (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
  (LABEL THREE)))
```

which will produce the following Free Menu:



And if the Free Menu were formatted as a Table, instead of in Rows, it would look like:



The following breakdown of the example explains how each part contributes to the Free Menu shown above.

`(PROPS FONT (MODERN 10))`

This line specifies the properties of the group that is the entire Free Menu. These properties are described in Section 28.7.4, Free Menu Group Properties. In this example, all items in the Free Menu, unless otherwise specified, will be in Modern 10.

`((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))`

This line of the Free Menu Description describes the first row of the menu. Since the `FORMAT` specification of a Free Menu is, by default, `ROW` formatting, this line sets the first row in the menu. If the menu were in `COLUMN` formatting, this position in the description would specify the first column in the menu.

In this example the first row contains only one item. The item is, by default, a type `MOMENTARY` item. It has its own Font declaration `(FONT (MODERN 10 BOLD))`, that overrides the font specified for the Free Menu as a whole, so the item appears bolded.

Finally, the item is justified, in this case centered. The `HJUSTIFY` Item Property indicates that the item is to be centered horizontally within its row.

`((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))`

This line specifies the second row of the menu. The second row has four very simple items, labeled NORTH, SOUTH, EAST, and WEST next to each other within the same row.

```
((PROPS ID ROW3 BOX 1)
 (LABEL ONE) (LABEL TWO) (LABEL THREE))
```

The third row in the menu is similar to the second row, except that it has a box drawn around it. The box is specified in the PROPS declaration for this row. Rows (and columns) are just like Groups in that the first thing in the declaration can be a list of properties for that row. In this case the row is named by giving it an ID property of ROW3. It is useful to name your groups if you want to be able to access and modify their properties later (via the function FM.GROUPPROP). It is boxed by specifying the BOX property with a value of 1, meaning draw the box one dot wide.

```
 ((PROPS ID ROW4)
  (LABEL ONE ID ALPHA)
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
   ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
         (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
         (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
        ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
          INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
         (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
  (LABEL THREE)))
```

This part of the description specifies the fourth row in the menu. This row consists of: an item labelled ONE, a group of items, and an item labelled THREE. That is, Free Menu thinks of the group as an entry, and formats the rest of the row just as it it were a large item.

```
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
   ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
    (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
    (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
   ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
     INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
    (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
```

The second part of this row is a nested group of items. It is declared as a group by placing the keyword GROUP as the first word in the declaration. A group can be declared anywhere a Free Menu Description can take a Free Menu Item Description (as opposed to a row or column declaration).

The first thing in what would have been the second item declaration in this row is the keyword GROUP. Following this keyword comes a normal group description, starting with an optional list of properties, and followed by any number of things to go in the group (based on the format of the group).

This group's Props declaration is:

```
(PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4).
```

It specifies that the group is to be formatted as a number of columns (instead of rows, the default). The entire group will have a background shade of 23130, and a box of width 2 around it, as you can see in the sample menu. The BOXSPACE declaration tells Free Menu to leave an extra four dots of room between the edge of the group (ie the box around the group) and the items in the group.

The first column of this group is a Collection of NWAY items:

```
((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
 (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
 (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
```

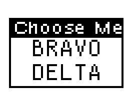
The three items, labelled A, B, and C are all declared as NWAY items, and are also specified to belong to the same NWAY Collection, Col1. This is how a number of NWAY items are collected together. The property NWAYPROPS (DESELECT T) on the first NWAY item specifies that the Col1 Collection is to have the Deselect property enabled. This simply means that the NWAY collection can be put in the state where none of the items (A, B, or C) are selected (highlighted). Additionally, each item is declared with a box whose width is one dot (pixel) around it.

The second column in this nested group is specified by:

```
((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
     INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
 (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35))
```

Column two contains two items, a STATE item and a DISPLAY item. The STATE item is labelled "Choose Me." A Label can be a string or a bitmap, as well as an atom. Selecting the STATE item will cause a pop-up menu to appear with two choices for the state of the item, BRAVO and DELTA. The items to go in the pop-up menu are designated by the MENUITEMS property.

The pop-up menu would look like:



The initial state of the "Choose Me" item is designated to be DELTA by the INITSTATE Item Property. The initial state can be anything; it does not have to be one of the items in the pop-up menu.

Next, the STATE item is Linked to a DISPLAY item, so that the current state of the item will be displayed in the Free Menu. The link's name is DISPLAY (a special link name for

STATE items), and the item linked to is described by the Link Description, (GROUP ALPHA). Normally the linked item can just be described by its ID. But in this case, there is more than one item whose ID is ALPHA (for the sake of this example), specifically the first item in the fourth row and the display item in this nested group. The form (GROUP ALPHA) tells Free Menu to search for an item whose ID is ALPHA, limiting the search to the items that are within this lexical group. The lexical group is the smallest group that is declared with the GROUP keyword (i.e., not row and column groups) that contains this item declaration. So in this case, Free Menu will link the STATE item to the DISPLAY item, rather than the first item in the fourth row, since *that* item is outside of the nested group. For further discussion of linking items, see Section 28.7.12, Free Menu Item Links.

Now, establish the DISPLAY item:

```
(TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)
```

We have given it the ID of Alpha that the above STATE item uses in finding the proper DISPLAY item to link to. This display item is used to display the current state of the item "Choose Me." Every item is required to have a Label property specified, but the label for this DISPLAY item will depend on the state of "Choose Me." That is, when the state of the "Choose Me" item is changed from DELTA to BRAVO, the label of the DISPLAY item will also change. The null string serves to hold the place for the changeable label.

A box is specified for this item. Since the label is the empty string, Free Menu would draw a very small box. Instead, the MAXWIDTH property indicates that the label, whatever it becomes, will be limited to a stringwidth of 35. The width restriction of 35 was chosen because it is big enough for each of the possible labels for this display item. So Free Menu draws the box big enough to enclose any item within this width restriction.

Finally we specify the final item in row four:

```
(LABEL THREE)
```

Free Menu Group Properties

Each group has properties. Most group properties are relevant and should be set in the group's PROPS list in the Free Menu Description. User properties can be freely included in the PROPS list. A few other properties are set up by the formatter. The macros FM.GROUPPROP or FM.MENUPROP allow access to group properties after the Free Menu is created.

ID — The identifier of this group. Setting the group ID is desirable, for example, if the application needs to get handles on items in particular groups, or access group properties.

FORMAT — One of ROW, COLUMN, TABLE, or EXPLICIT. The default is ROW.

FONT — A font description of the form (FAMILY SIZE FACE), or a FONTDESCRIPTOR data type. This will be the default font for each item in this group. The default font of the top group is the value of the variable DEFAULTFONT.

COORDINATES One of GROUP or MENU. This property applies only to EXPLICIT formatting. If GROUP, the items in the EXPLICIT group are positioned in coordinates relative to the lower left corner of the group, as determined by the mother group. If MENU, which is the default, the items are positioned relative to the lower left corner of the menu.

LEFT Specifies a left offset for this group, pushing the group to the right.

BOTTOM Specifies a bottom offset for this group, pushing the group up.

ROWSPACE Specifies the number of dots between rows in this group.

COLUMNSPACE Specifies the number of dots between columns in this group.

BOX Specifies the number of dots in the box around this group of items.

BOXSHADE Specifies the shade of the box.

BOXSPACE Specifies the number of bits between the box and the items.

BACKGROUND The background shade of this group. Nested groups inherit this background shade, but items in this group and nested groups do not. This is because, in general, it is difficult to read text on a background, so items appear on a white background by default. This can be overridden by the BACKGROUND Item Property.

Other Group Properties

The following group properties are set up and maintained by Free Menu. The application should probably not change any of these properties.

ITEMS A list of the items in the group.

REGION The region that is the extent of the items in the group.

MOTHER The ID of the group that is the mother of this group.

DAUGHTERS A list of ID of groups which are daughters to this group.

Free Menu Items

Each Free Menu Item is stored as an instance of the data type FREEMENUITEM. Free Menu Items can be thought of as objects, each item having its own particular properties, such as its type, label, and mouse event functions. A number of useful item types, described in Section 28.7.11, Predefined Item Types, are predefined by Free Menu. New types of items can be defined by the application, using

Display items as a base. Each Free Menu Item is created from a Free Menu Item Description when the Free Menu is created.

> CAUTION: Edit (and thus Number) Freemenu Items do not perform well when boxed or when there is another item to the right in the same row. The display to the right of the edit item may be corrupted under editing and fm.changelabel operations.

Free Menu Item Descriptions

A Free Menu Item Description is a list in property list format, specifying the properties of the item. For example:

```
(LABEL Refetch SELECTEDFN MY.REFETCHFN)
```

describes a MOMENTARY item labelled Refetch, with the function MY.REFETCHFN to be called when the item is selected. None of the property values in an item description are evaluated. When constructing Free Menu descriptions that incorporate evaluated expressions (for example labels that are bitmaps) it is helpful to use the backquote facility. For instance, if the value of the variable MYBITMAP is a bitmap, then

```
(FREEMENU `(((LABEL A) (LABEL ,MYBITMAP))))
```

would create a Free Menu of one row, with two items in that row, the second of which has the value of MYBITMAP as its label.

Free Menu Item Properties

The following Free Menu Item Properties can be set in the Item Description. Any other properties given in an Item Description will be treated as user properties, and will be saved on the USERDATA property of the item.

TYPE — The type of the item. Choose from one of the Free Menu Item type keywords MOMENTARY, TOGGLE, 3STATE, STATE, NWAY, EDITSTART, EDIT, NUMBER, or DISPLAY. The default is MOMENTARY.

LABEL — An atom, string, or bitmap. Bitmaps are always copied, so that the original will not be changed. This property must be specified for every item.

FONT — The font in which the item appears. The default is the font specified for the group containing this item. Can be a font description of the form (FAMILY SIZE FACE), or a FONTDESCRIPTOR data type.

ID — May be used to specify a unique identifier for this item, but is not necessary.

LEFT and BOTTOM — When ROW, COLUMN, or TABLE formatting, these specify offsets, pushing the item right and up, respectively, from where the formatter would have put the item. In EXPLICIT formatting, these are the actual coordinates of the item, in the coordinate system given by the group's COORDINATES property.

HJUSTIFY — Indicates horizontal justification type: LEFT, CENTER, or RIGHT. Specifies that this item is to be horizontally justified within the extent of its group. Note that the main group, as opposed to the smaller row or column group, is used.

VJUSTIFY — Specifies that this item is to be vertically justified . Values are TOP, MIDDLE, or BOTTOM.

HIGHLIGHT — Specifies the highlighted looks of the item, that is, how the item changes when a mouse event occurs on it. See Section 28.7.12, Free Menu Item Highlighting, for more details on highlighting.

MESSAGE — Specifies a string that will be printed in the prompt window after a mouse cursor selects this item for MENUHELDWAIT milliseconds. Or, if an atom, treated as a function to get the message. The function is passed three arguments, ITEM, WINDOW, and BUTTONS, and should return a string. The default is a message appropriate to the type of the item.

INITSTATE — Specifies the initial state of the item. This is only appropriate to TOGGLE, 3STATE, and STATE items.

MAXWIDTH — Specifies the width allowed for this item. The formatter will leave enough space after the item for the item to grow to this width without collisions.

MAXHEIGHT — Similar to MAXWIDTH, but in the vertical dimension.

BOX — Specifies the number of bits in the box around this item. Boxes are made around MAXWIDTH and MAXHEIGHT dimensions. If unspecified, no box is drawn.

BOXSHADE — Specifies the shade that the box is drawn in. The default is BLACKSHADE.

BOXSPACE — Specifies the number of bits between the box and the label. The default is one bit.

BACKGROUND — Specifies the background shade on which the item appears. The default is WHITESHADE, regardless of the group's background.

LINKS — Can be used to link this item to other items in the Free Menu.

Mouse Properties

The following properties provide a way for application functions to be called under certain mouse events. These functions are called with the ITEM, the WINDOW, and the BUTTONS passed as arguments. These application functions do not interfere with any Free Menu system functions that take care of handling the different item types. In each case, though, the application function is called after the system function. The default for all of these functions is NILL. The value of each of the following properties can be the name of a function, or a lambda expression.

SELECTEDFN Specifies the function to be called when this item is selected. The Edit and EditStart items cannot have a SELECTEDFN. See the Edit Free Menu item description in Section 28.7.11, Predefined Item Types, for more information.

DOWNFN Specifies the function to be called when the item is selected with the mouse cursor.

HELDFN Specifies the function to be called repeatedly when the item is selected with the mouse cursor.

MOVEDFN Specifies the function to be called when the mouse cursor moves off this item (mouse buttons are still depressed).

System Properties

The following Free Menu Item properties are set and maintained by Free Menu. The application should probably not change these properties directly.

GROUPID Specifies the ID of the smallest group that the item is in. For example, in a row formatted group, the item's GROUPID will be set to the ID of the row that the item is in, not the ID of the whole group.

STATE Specifies the current state of TOGGLE, 3STATE, or STATE items. The state of an NWAY item behaves like that of a toggle item.

BITMAP Specifies the bitmap from which the item is displayed.

REGION Specifies the region of the item, in window coordinates. This is used for locating the display position, as well as determining the mouse sensitive region of the item.

MAXREGION Specifies the maximum region the item may occupy, determined by the MAXWIDTH and MAXHEIGHT properties (see Section 28.7.8, Free Menu item Properties). This is used by the formatter and the display routines.

SYSDOWNFN

SYSMOVEDFN

SYSSELECTEDFN These are the system mouse event functions, set up by Free Menu according to the item type. These functions are called before the mouse event functions, and are used to implement highlighting, state changes, editing, etc.

USERDATA Specifies how any other properties are stored on this list in property list format. This list should probably not need to be manipulated directly.

Predefined Item Types

MOMENTARY [Free Menu Item]

MOMENTARY items are like command buttons. When the button is selected, its associated function is called.

TOGGLE [Free Menu Item]

Toggle items are simple two-state buttons. When pressed, the button is highlighted; it stays that way until pressed again. The states of a toggle button are T and NIL; the initial state is NIL.

3STATE [Free Menu Item]

3STATE items rotate through NIL, T, and OFF, states each time they are pressed. The default looks of the OFF state are with a diagonal line through the button, while T is highlighted, and NIL is normal. The default initial state is NIL.

The following Item Property applies to 3STATE items:

OFF Specifies the looks of a 3STATE item in its OFF state. Similar to HIGHLIGHT. The default is that the label gets a diagonal slash through it.

NOTE: If you specify special highlighting (a different bitmap of string) for Toggle or 3State items AND use this item in a group formatted as a Column or a Table, the highlight looks of the item may not appear in the correct place.

STATE [Free Menu Item]

STATE items are general multiple state items. The following Item Property determines how the item changes state:

CHANGESTATE This Item Property can be changed at any time to change the effect of the item. If a MENU data type, this menu pops up when the item is selected, and the user can select the new state. Otherwise, if this property is given, it is treated as a function name, which is passed three arguments, ITEM, WINDOW, and BUTTONS. This function can do whatever it wants, and is expected to return the new state (an atom, string, or bitmap), or NIL, indicating the state should not change. The state of the item can automatically be indicated in the Free Menu, by setting up a DISPLAY link to a DISPLAY item in the menu (see Section 28.7.13, Free Menu Item Links). If such a link exists, the label of the DISPLAY item will be changed to the new state. The possible states are not restricted at all, with the exception of selections from a pop-up menu. The state can be changed to any atom, string, or bitmap, manually via FM.CHANGESTATE.

The following Item Properties are relevant to STATE items when building a Free Menu:

MENUITEMS If specified, should be a list of items to go in a pop-up menu for this item. Free Menu will build the menu and save it as the CHANGESTATE property of the item.

MENUFONT The font of the items in the pop-up menu.

MENUTITLE The title of the pop-up menu. The default title is the label of the STATE item.

NWAY [Free Menu Item]

NWAY items provide a way to collect any number of items together, in any format within the Free Menu. Only one item from each Collection can be selected at a time, and that item is highlighted to indicate this. The following Item Properties are particular to NWAY items:

COLLECTION An identifier that specifies which NWAY Collection this item belongs to.

NWAYPROPS A property list of information to be associated with this collection. This property is only noticed in the Free Menu Description on the first item in a COLLECTION. NWAY Collections are formed by creating a number of NWAY items with the same COLLECTION property. Each NWAY item acts individually as a Toggle item, and can have its own mouse event functions. Each NWAY Collection itself has properties, its state for instance. After the Free Menu is created, these Collection properties can be accessed by the macro FM.NWAYPROPS. Note that NWAY Collections are different from Free Menu Groups. There are three NWAY Collection properties that Free Menu looks at:

DESELECT If given, specifies that the Collection can be deselected, yielding a state in which no item in the Collection is selected. When this property is set, the Collection can be deselected by selecting any item in the Collection and pressing the right mouse button .

STATE The current state of the Collection, which is the actual item selected.

INITSTATE Specifies the initial state of the Collection. The value of this property is an Item Link Description

EDIT [Free Menu Item]

EDIT items are textual items that can be edited. The label for an EDIT item cannot be a bitmap. When the item is selected an edit caret appears at that cursor position within the item, allowing insertion and deletion of characters at that point. If selected with the right mouse button, the item is cleared before editing starts. While editing, the left mouse button moves the caret to a new position within the item. The right mouse button deletes from the caret to the cursor. CONTROL-W deletes the previous word. Editing is stopped when another item is selected, when the user moves the cursor into another TTY window and clicks the cursor, or when the Free Menu function FM.ENDEDIT is called (called when the Free Menu is reset, or the window is closed). The Free Menu editor will time out after about a minute, returning automatically. Because of the many ways in which editing can terminate, EDIT items are not allowed to have a SELECTEDFN, as it is not clear when this function should be called. Each EDIT item should have an ID specified, which is used when getting the state of the Free Menu, since the string being edited is defined as the state of the item, and thus cannot distinguish edit items. The following Item Properties are specific to EDIT items.

MAXWIDTH Specifies the maximum string width of the item, in bits, after which input will be ignored. If MAXWIDTH is not specified, the items becomes infinitely wide and input is never restricted.

INFINITEWIDTH This property is set automatically when MAXWIDTH is not specified. This tells Free Menu that the item has no right end, so that the item becomes mouse sensitive from its left edge to the right edge of the window, within the vertical space of the item.

In Medley, Changestate of an infinite width Edit item to a smaller item clears the old item properly.

LIMITCHARS The input characters allowed can be restricted in two ways: If this item property is a list, it is treated as a list of legal characters; any character not in the list will be ignored. If it is an atom, it is treated as the name of a test predicate, which is passed three arguments, ITEM, WINDOW, and CHARACTER, when each character is typed. This predicate should return T if the character is legal, NIL otherwise. The LIMITCHARS function can also call FM.ENDEDIT to force the editor to terminate, or FM.SKIPNEXT, to cause the editor to jump to the next edit item in the menu.

ECHOCHAR This item property can be set to any character. This character will be echoed in the window, regardless of what character is typed. However the item's label contains the actual string typed. This is useful for operations like password prompting. If ECHOCHAR is used, the font of the item must be fixed pitch. Unrestricted EDIT items should not have other items to their right in the menu, as they will be replaced. If the item is boxed, input is restricted to what will fit in the box. Typing off the edge of the window will cause the window to scroll appropriately. Control characters can be edited, including the carriage return and line feed, and they are echoed as a black box. While editing, the Skip / Next key ends editing the current item, and starts editing the next EDIT item in the Free Menu.

NUMBER [Free Menu Item]

NUMBER items are EDIT items that are restricted to numerals. The state of the item is coerced to the the number itself, not a string of numerals. There is one NUMBER- specific Item Property:

NUMBERTYPE If FLOATP (or FLOAT), then decimals are accepted. Otherwise only whole numbers can be edited.

EDITSTART [Free Menu Item]

EDITSTART items serve the purpose of starting editing on another item when they are selected. The associated Edit item is linked to the EditStart item by an EDIT link (see Free Menu Item Links below). If the EDITSTART item is selected with the right mouse button, the Edit item is cleared before editing is started. Similar to EDIT items, EDITSTART items cannot have a SELECTEDFN, as it is not clear when the associated editing will terminate.

In Medley, EDITSTART items linked to a Number item properly set number state when editing has completed.

DISPLAY [Free Menu Item]

DISPLAY items serve two purposes. First, they simply provide a way of putting dummy text in a Free Menu, which does nothing when selected. The item's label can be changed, though. Secondly, DISPLAY items can be used as the base for new item types. The application can create new item types by specifying DOWNFN, HELDFN, MOVEDFN, and SELECTEDFN for a DISPLAY item, making it behave as desired.

Free Menu Item Highlighting

Each Free Menu Item can specify how it wants to be highlighted. First of all, if the item does not specify a HIGHLIGHT property, there are two default highlights. If the item is not boxed, the label is simply inverted, as in normal menus. If the item is boxed, it is highlighted in the shade of the box. Alternatively, the value of the HIGHLIGHT property can be a SHADE, which will be painted on top of the item when a mouse event occurs on it. Or the HIGHLIGHT property can be an alternate label, which can be an atom, string or bitmap. If the highlight label is a different size than the item label, the formatter will leave enough space for the larger of the two. In all of these cases, the looks of the highlighted item are determined when the Free Menu is built, and a bitmap of the item with these looks is created. This bitmap is stored on the item's HIGHLIGHT property, and simply displayed when a mouse event occurs. The value of the highlight property in the Item Description is copied to the USERDATA list, in case it is needed later for a label change.

Free Menu Item Links

Links between items are useful for grouping items in abstract ways. In particular, links are used for associating EDITSTART items with their item to edit, and STATE items with their state display. The Free Menu Item property LINKS is a property list, where the value of each Link Name property is a pointer to another item. In the Item Description, the value of the LINK property should be a property list as above. The value of each Link Name property is a Link Description. A Link Description can be one of the following forms:

<ID> An ID of an item in the Free Menu. This is acceptable if items can be distinguished by ID alone.

(<GROUPID> <ID>) A list whose first element is a GROUPID, and whose second element is the ID of an item in that group. This way items with similar purposes, and thus similar ID's, can be distinguished across groups.

(GROUP <ID>) A list whose first element is the keyword GROUP, and whose second element is an item ID. This form describes an item with ID, in the same group that this item is in. This way you do not need to know the GROUPID, just which group it is in.

Then after the entire menu is built, the links are set up, turning the Link Descriptions into actual pointers to Free Menu Items. There is no reason why circular Item Links cannot be created, although such a link would probably not be very useful. If circular links are created, the Free Menu will not be garbage collected after it is not longer being used. The application is responsible for breaking any such links that it creates.

Free Menu Window Properties

`FM.PROMPTWINDOW` Specifies the window that Free Menu should use for displaying the item's messages. If not specified, `PROMPTWINDOW` is used.

`FM.BACKGROUND` The background shade of the entire Free Menu. This property can be set automatically by specifying a `BACKGROUND` argument to the function `FREEMENU`. The window border must be 4 or greater when a Free Menu background is used, due to the way the Window System handles window borders.

`FM.DONTRESHAPE` Normally, Free Menu will attempt to use empty space in a window by pushing items around to fill the space. When a Free Menu window is reshaped, the items are repositioned in the new shape. This can be disabled by setting the `FM.DONTRESHAPE` window property.

Free Menu Interface Functions

(**FREEMENU** *DESCRIPTION TITLE BACKGROUND BORDER*) [Function]

Creates a Free Menu from a Free Menu Description, returning the window. This function will return quickly unless new display fonts have to be created.

Accessing Functions

(**FM.GETITEM** *ID GROUP WINDOW*) [Function]

Gets item *ID* in *GROUP* of the Free Menu in *WINDOW*. This function will search the Free Menu for an item whose *ID* property matches, or secondly whose `LABEL` property matches *ID*. If *GROUP* is `NIL`, then the entire Free Menu is searched. If no matching item is found, `NIL` is returned.

(**FM.GETSTATE** *WINDOW*) [Function]

Returns in property list format the ID and current `STATE` of every `NWAY` Collection and item in the Free Menu. If an item's or Collection's state is `NIL`, then it is not included in the list. This provides an easy way of getting the state of the menu all at once. If the state of only one item or Collection is needed, the application can directly access the `STATE` property of that object using the Accessing Macros described in Section 28.7.20, Free Menu Macros. This function can be called when editing is in progress, in which case it will provide the label of the item being edited at that point.

Changing Free Menus

Many of the following functions operate on Free Menu Items, and thus take the item as an argument. The *ITEM* argument to these functions can be the Free Menu Item itself, or just a reference to the item. In the second case, FM.GETITEM (see Section 28.7.16, Accessing Functions) will be used to find the item in the Free Menu. The reference can be in one of the following forms:

<ID> Specifies the first item in the Free Menu whose ID or LABEL property matches <ID>.

(<GROUPID> <ID>) Specifies the item whose ID or LABEL property matches <ID> within the group specified by <GROUPID>.

(**FM.CHANGELABEL** *ITEM NEWLABEL WINDOW UPDATEFLG*) [Function]

Changes an *ITEM*'s label after the Free Menu has been created. It works for any type of item, and STATE items will remain in their current state. If the window is open, the item will be redisplayed with its new appearance. *NEWLABEL* can be an atom, a string, or a bitmap (except for EDIT items), and will be restricted in size by the MAXWIDTH and MAXHEIGHT Item Properties. If these properties are unspecified, the *ITEM* will be able to grow to any size. *UPDATEFLG* specifies whether or not the regions of the groups in the menu are recalculated to take into account the change of size of this item. The application should not change the label of an EDIT item while it is being edited. The following Item Property is relevant to changing labels:

CHANGELABELUPDATE Exactly like UPDATEFLG except specified on the item, rather than as a function paramater.

(**FM.CHANGESTATE** *X NEWSTATE WINDOW*) [Function]

Programmatically changes the state of items and NWAY Collections. *X* is either an item or a Collection name. For items *NEWSTATE* is a state appropriate to the type of the item. For NWAY Collections, *NEWSTATE* should be the desired item in the Collection, or NIL to deselect. For EDIT and NUMBER items, this function just does a label change. If the window is open, the item will be redisplayed.

(**FM.RESETSTATE** *ITEM WINDOW*) [Function]

Sets an *ITEM* back to its initial state.

(**FM.RESETMENU** *WINDOW*) [Function]

Resets every item in the menu back to its initial state.

(**FM.RESETSHAPE** *WINDOW ALWAYSFLG*) [Function]

Reshapes the *WINDOW* to its full extent, leaving the lower-left corner unmoved. Unless *ALWAYSFLG* is T, the window will only be increased in size as a result of resetting the shape.

(**FM.RESETGROUPS** *WINDOW*) [Function]

Recalculates the extent of each group in the menu, updating group boxes and backgrounds appropriately.

(**FM.HIGHLIGHTITEM** *ITEM WINDOW*) [Function]

Programmatically forces an *ITEM* to be highlighted. This might be useful for *ITEM*s which have a direct effect on other *ITEM*s in the menu. The *ITEM* will be highlighted according to its HIGHLIGHT property, as described in Section 28.7.12, Free Menu Item Highlighting. This highlight is temporary, and will be lost if the *ITEM* is redisplayed, by scrolling for example.

Editor Functions

(**FM.EDITITEM** *ITEM WINDOW CLEARFLG*) [Function]

Starts editing an EDIT or NUMBER *ITEM* at the beginning of the *ITEM*, as long as the *WINDOW* is open. This function will most likely be useful for starting editing of an *ITEM* that is currently the null string. If *CLEARFLG* is set, the *ITEM* is cleared first.

(**FM.SKIPNEXT** *WINDOW CLEARFLG*) [Function]

Causes the editor to jump to the beginning of the next EDIT item in the Free Menu. If *CLEARFLG* is set, then the next item will be cleared first. If there is not another EDIT item in the menu, this function will simply cause editing to stop. If this function is called when editing is not in progress, editing will begin on the first EDIT item in the menu. This function can be called from any process, and can also be called from inside the editor, in a LIMITCHARS function.

(**FM.ENDEDIT** *WINDOW WAITFLG*) [Function]

Stops any editing going on in *WINDOW*. If *WAITFLG* is T, then block until the editor has completely finished. This function can be called from another process, or from a LIMITCHARS function.

(**FM.EDITP** *WINDOW*) [Function]

If an item is in the process of being edited in the Free Menu *WINDOW*, that item is returned. Otherwise, NIL is returned.

Miscellaneous Functions

(**FM.REDISPLAYMENU** *WINDOW*) [Function]

Redisplays the entire Free Menu in its *WINDOW*, if the *WINDOW* is open.

(**FM.REDISPLAYITEM** *ITEM WINDOW*) [Function]

Redisplays a particular Free Menu *ITEM* in its *WINDOW*, if the *WINDOW* is open.

(**FM.SHADE** *X SHADE WINDOW*) [Function]

X can be an item, or a group ID. *SHADE* is painted on top of the item or group. Note that this is a temporary operation, and will be undone by redisplaying. For more permanent shading, the application may be able to add a REDEDISPLAYFN and SCROLLFN for the window as necessary to update the shading.

(**FM.WHICHITEM** *WINDOW POSorX Y*) [Function]

Locates and identifies an item from its known location within the *WINDOW*. If *WINDOW* is NIL, (WHICHW) is used, and if *POSorX* is NIL, the current cursor location is used.

(**FM.TOPGROUPID** *WINDOW*) [Function]

Returns the ID of the top group of this Free Menu.

Free Menu Macros

These Accessing Macros are provided to allow the application to get and set information in the Free Menu data structures. They are implemented as macros so that the operation will compile into the actual access form, rather than figuring that out at run time.

(**FM.ITEMPROP** *ITEM PROP {VALUE}*) [Macro]

Similar to WINDOWPROP, this macro provides an easy access to the fields of a Free Menu Item. The function FM.GETITEM gets the *ITEM*, described in Section 28.7.16, Accessing Function. *VALUE* is optional, and if not given, the current value of the *PROP* property will be returned. If *VALUE* is given, it will be used as the new value for that *PROP*, and the old value will be returned. When a call to FM.ITEMPROP is compiled, if the *PROP* is known (quoted in the calling form), the macro figures out what field to access, and the appropriate Data Type access form is compiled. However, if the *PROP* is not known at compile time, the function FM.ITEMPROP, which goes through the necessary property selection at run time, is compiled. The TYPE and USERDATA properties of a Free Menu Item are Read Only, and an error will result from trying to change the value of one of these properties.

(**FM.GROUPPROP** *WINDOW GROUP PROP {VALUE}*) [Macro]

Provides access to the Group Properties set up in the *PROPS* list for each group in the Free Menu Description. *GROUP* specifies the ID of the desired group, and *PROP* the name of the desired property. If *VALUE* is specified, it will become the new value of the property, and the old value will be returned. Otherwise, the current value is returned.

(**FM.MENUPROP** *WINDOW PROP {VALUE}*) [Macro]

Provides access to the group properties of the top-most group in the Free Menu, that is to say, the entire menu. This provides an easy way for the application to attach properties to the menu as a whole, as well as access the Group Properties for the entire menu.

(**FM.NWAYPROP** *WINDOW COLLECTION PROP {VALUE}*) [Macro]

This macro works just like FM.GROUPPROP, except it provides access to the NWay Collections.

Attached Windows

The attached window facility makes it easy to manipulate a group of window as a unit. Standard window operations like moving, reshaping, opening, and closing can be done so that it appears to the user as if the windows are a single entity. Each collection of attached windows has one main window and any number of other windows that are "attached" to it. Moving or reshaping the main window causes all of the attached windows to be moved or reshaped as well. Moving or reshaping an attached window does not affect the main window.

Attached windows can have other windows attached to them. Thus, it is possible to attach window A to window B when B is already attached to window C. Similarly, if A has other windows attached to it, it can still be attached to B.

(**ATTACHWINDOW** *WINDOWTOATTACH MAINWINDOW EDGE POSITIONONEDGE WINDOWCOMACTION*) [Function]

Associates *WINDOWTOATTACH* with *MAINWINDOW* so that window operations done to *MAINWINDOW* are also done to *WINDOWTOATTACH* (the exact set of window operations passed between main windows and attached windows is described in the Window Operations and Attached Windows section below). ATTACHWINDOW moves *WINDOWTOATTACH* to the correct position relative to *MAINWINDOW*.

Note: A window can be attached to only one other window. Attaching a window to a second window will detach it from the first. Attachments can not form loops. That is, a window cannot be attached to itself or to a window that is attached to it. ATTACHWINDOW will generate an error if this is attempted.

EDGE determines which edge of *MAINWINDOW* the attached window is positioned along: it should be one of TOP, BOTTOM, LEFT, or RIGHT. If *EDGE* is NIL, it defaults to TOP.

POSITIONONEDGE determines where along *EDGE* the attached window is positioned. It should be one of the following:

LEFT — The attached window is placed on the left (of a TOP or BOTTOM edge).

RIGHT — The attached window is placed on the right (of a TOP or BOTTOM edge).

BOTTOM — The attached window is placed on the bottom (of a LEFT or RIGHT edge).

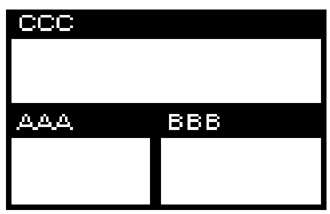
TOP — The attached window is placed on the top (of a LEFT or RIGHT edge).

CENTER — The attached window is placed in the center of the edge.

JUSTIFY

or NIL — The attached window is placed to fill the entire edge. ATTACHWINDOW reshapes the window if necessary.

Note: The width or height used to justify an attached window includes any other windows that have already been attached to *MAINWINDOW*. Thus (ATTACHWINDOW BBB AAA 'RIGHT 'JUSTIFY) followed by (ATTACHWINDOW CCC AAA 'TOP 'JUSTIFY) will put CCC across the top of both BBB and AAA:



WINDOWCOMACTION provides a convenient way of specifying how *WINDOWTOATTACH* responds to right button menu commands. The window property PASSTOMAINCOMS determines which right button menu commands are directly applied to the attached window, and which are passed to the main window (see the Window Operations and Attached Windows section below). Depending on the value of *WINDOWCOMACTION*, the PASSTOMAINCOMS window property of *WINDOWTOATTACH* is set as follows:

NIL — PASSTOMAINCOMS is set to (CLOSEW MOVEW SHAPEW SHRINKW BURYW), so right button menu commands to close, move, shape, shrink, and bury are passed to the main window, and all others are applied to the attached window.

LOCALCLOSE — PASSTOMAINCOMS is set to (MOVEW SHAPEW SHRINKW BURYW), which is the same as when *WINDOWCOMACTION* is NIL, except that the attached window can be closed independently.

HERE — PASSTOMAINCOMS is set to NIL, so all right button menu commands are applied to the attached window.

MAIN — PASSTOMAINCOMS is set to T, so all right button menu commands are passed to the main window.

Note: If the user wants to set the PASSTOMAINCOMS window property of an attached window to something else, it must be done after the window is attached, since ATTACHWINDOW modifies this window property.

(**DETACHWINDOW** *WINDOWTODETACH*) [Function]

Detaches *WINDOWTODETACH* from its main window. Returns a dotted pair (EDGE . POSITIONONEDGE) if *WINDOWTODETACH* was an attached window, NIL otherwise. This does not close *WINDOWTODETACH*.

(**DETACHALLWINDOWS** *MAINWINDOW*) [Function]

Detaches and closes all windows attached to *MAINWINDOW*.

(**FREEATTACHEDWINDOW** *WINDOW*) [Function]

Detaches the attached window *WINDOW*. In addition, other attached windows above (in the case of a TOP attached window) or below (in the case of a BOTTOM attached window) are moved closer to the main window to fill the gap.

Note: Attached windows that "reject" the move operation (see REJECTMAINCOMS below) are not moved.

Note: FREEATTACHEDWINDOW currently doesn't handle LEFT or RIGHT attached windows.

(**REMOVEWINDOW** *WINDOW*) [Function]

Closes *WINDOW*, and calls FREEATTACHEDWINDOW to move other attached windows to fill any gaps.

(**REPOSITIONATTACHEDWINDOWS** *WINDOW*) [Function]

Repositions every window attached to *WINDOW*, in the order that they were attached. This is useful as a RESHAPEFN for main windows with attached window that don't want to be reshaped, but do want to keep their position relative to the main window when the main window is reshaped.

Note: Attached windows that "reject" the move operation (see REJECTMAINCOMS below) are not moved.

(**MAINWINDOW** *WINDOW RECURSEFLG*) [Function]

If *WINDOW* is not a window, it generates an error. If *WINDOW* is closed, it returns *WINDOW*. If *WINDOW* is not attached to another window, it returns *WINDOW* itself. If *RECURSEFLG* is NIL and *WINDOW* is attached to a window, it returns that window. If *RECURSEFLG* is T, it returns the first window up the "main window" chain starting at *WINDOW* that is not attached to any other window.

(**ATTACHEDWINDOWS** *WINDOW COM*) [Function]

Returns the list of windows attached to *WINDOW*.

If *COM* is non-NIL, only those windows attached to *WINDOW* that do not reject the window operation *COM* are returned (see REJECTMAINCOMS).

(**ALLATTACHEDWINDOWS** *WINDOW*) [Function]

Returns a list of all of the windows attached to *WINDOW* or attached to a window attached to it.

(**WINDOWREGION** *WINDOW COM*) [Function]

Returns the screen region occupied by *WINDOW* and its attached windows, if it has any.

If *COM* is non-NIL, only those windows attached to *WINDOW* that do not reject the window operation *COM* are considered in the calculation (see REJECTMAINCOMS).

(**WINDOWSIZE** *WINDOW*) [Function]

Returns the size of *WINDOW* and its attached windows (if any), as a dotted pair (WIDTH . HEIGHT).

(**MINATTACHEDWINDOWEXTENT** *WINDOW*) [Function]

Returns the minimum size that *WINDOW* and its attached windows (if any) will accept, as a dotted pair (WIDTH . HEIGHT).

Attaching Menus To Windows

The following functions are provided to associate menus to windows.

(**MENUWINDOW** *MENU VERTFLG*) [Function]

Returns a closed window that has the menu *MENU* in it. If *MENU* is a list, a menu is created with *MENU* as its ITEMS menu field. Otherwise, *MENU* should be a menu. The returned window has the appropriate RESHAPEFN, MINSIZE and MAXSIZE window properties to allow its use in a window group.

If both the MENUROWS and MENUCOLUMNS fields of *MENU* are NIL, *VERTFLG* is used to set the default menu shape. If *VERTFLG* is non-NIL, the MENUCOLUMNS field of *MENU* will be set to 1 (the menu items will be listed vertically); otherwise the MENUROWS field of *MENU* will be set to 1 (the menu items will be listed horizontally).

(**ATTACHMENU** *MENU MAINWINDOW EDGE POSITIONONEDGE NOOPENFLG*) [Function]

Creates a window that contains the menu *MENU* (by calling *MENUWINDOW*) and attaches it to the window *MAINWINDOW* on edge *EDGE* at position *POSITIONONEDGE*. The menu window is opened unless *MAINWINDOW* is closed, or *NOOPENFLG* is T.

If EDGE is either LEFT or RIGHT, MENUWINDOW will be called with VERTFLG = T, so the menu items will be listed vertically; otherwise the menu items will be listed horizontally. These defaults can be overridden by specifying the MENUROWS or MENUCOLUMNS fields in *MENU*.

(**CREATEMENUEDWINDOW** *MENU WINDOWTITLE LOCATION WINDOWSPEC*) [Function]

Creates a window with an attached menu and returns the main window. *MENU* is the only required argument, and may be a menu or a list of menu items. *WINDOWTITLE* is a string specifying the title of the main window. *LOCATION* specifies the edge on which to place the menu; the default is TOP. *WINDOWSPEC* is a region specifying a region for the aggregate window; if NIL, the user is prompted for a region.

Examples:

```
(SETQ MENUW
  (MENUWINDOW
     (create MENU
        ITEMS ← '(smaller LARGER)
        MENUFONT ← '(MODERN 12)
        TITLE ← "zoom controls"
        CENTERFLG ← T
        WHENSELECTEDFN ← (FUNCTION ZOOMMAINWINDOW))))
```

creates (but does not open) a menu window that contains the two items "smaller" and "LARGER" with the title "zoom controls" and that calls the function ZOOMMAINWINDOW when an item is selected. Note that the menu items will be listed horizontally, because MENUWINDOW is called with VERTFLG = NIL, and the menu does not specify either a MENUROWS or MENUCOLUMNS field.

```
(ATTACHWINDOW MENUW
   (CREATEW '(50 50 150 50))
   'TOP
   'JUSTIFY)
```

creates a window on the screen and attaches the above created menu window to its top:


```
(CREATEMENUEDWINDOW
   (create MENU
      ITEMS ← '(smaller LARGER)
      MENUFONT ← '(MODERN 12)
      TITLE ← "zoom controls"
      CENTERFLG ← T
      WHENSELECTEDFN ← (FUNCTION ZOOMMAINWINDOW))))
```

creates the same sort of window in one step, prompting the user for a region.

Attached Prompt Windows

Many packages have a need to display status information or prompt for small amounts of user input in a place outside their standard window. A convenient way to do this is to attach a small window to the top of the program's main window. The following functions do so in a uniform way that can be depended on among diverse applications.

(**GETPROMPTWINDOW** *MAINWINDOW #LINES FONT DONTCREATE*) [Function]

Returns the attached prompt window associated with *MAINWINDOW*, creating it if necessary. The window is always attached to the top of *MAINWINDOW*, has DSPSCROLL set to T, and has a PAGEFULLFN of NILL to inhibit page holding. The window is at least *#LINES* lines high (default 1); if a pre-existing window is shorter than that, it is reshaped to make it large enough. *FONT* is the font to give the prompt window (defaults to the font of *MAINWINDOW*), and applies only when the window is first created. If *DONTCREATE* is true, returns the window if it exists, otherwise NIL without creating any prompt window.

(**REMOVEPROMPTWINDOW** *MAINWINDOW*) [Function]

Detaches the attached prompt window associated with *MAINWINDOW* (if any), and closes it.

Window Operations And Attached Windows

When a window operation, such as moving or clearing, is performed on a window, there is a question about whether or not that operation should also be performed on the windows attached to it or performed on the window it is attached to. The "right" thing to do depends on the window operation: it makes sense to independently redisplay a single window in a collection of windows, whereas moving a single window usually implies moving the whole group of windows. The interpretation of window operations also depends on the application that the window group is used for. For some applications, it may be desirable to have a window group where individual windows can be moved away from the group, but still be conceptually attached to the group for other operations. The attached window facility is flexible enough to allow all of these possibilities.

The operation of window operations can be specified by each attached window, by setting the following two window properties:

PASSTOMAINCOMS [Window Property]

Value is a list of window commands (e.g. CLOSEW, MOVEW) which, when selected from the attached window's right-button menu, are actually applied to the central window in the group, instead of being applied to the attached window itself. The "central window" is the first window up the "main window" chain that is not attached to any other window.

If PASSTOMAINCOMS is NIL, all window operations are directly applied to the attached window. If PASSTOMAINCOMS is T, all window operations are passed to the central window.

Note: ATTACHWINDOW allows this window property to be set to commonly-used values by using its WINDOWCOMACTION argument. ATTACHWINDOW always sets this window property, so users must modify it directly only after attaching the window to another window.

REJECTMAINCOMS [Window Property]

Value is a list of window commands that the attached window will not allow the main window to apply to it. This is how a window can say "leave me out of this group operation."

If REJECTMAINCOMS is NIL, all window commands may be applied to this attached window. If REJECTMAINCOMS is T, no window commands may be applied to this attached window.

The PASSTOMAINCOMS and REJECTMAINCOMS window properties affect right-button menu operations applied to main windows or attached windows, and the action of programmatic window functions (SHAPEW, MOVEW, etc.) applied to main windows. However, these window properties do not affect the action of window functions applied to attached windows.

The following list describes the behavior of main and attached windows under the window operations, assuming that all attached windows have their REJECTMAINCOMS window property set to NIL and PASSTOMAINCOMS set to (CLOSEW MOVEW SHAPEW SHRINKW BURYW) (the default if ATTACHWINDOW is called with WINDOWCOMACTION = NIL).

The behavior for any particular operation can be changed for particular attached windows by setting the standard window properties (e.g., MOVEFN or CLOSEFN) of the attached window. An exception is the TOTOPFN property of an attached window, that is set to bring the whole window group to the top and should not be set by the user (although users can add functions to the TOTOPFN window property).

Move — If the main window moves, all attached windows move with it, and the relative positioning between the main window and the attached windows is maintained. If the region is determined interactively, the prompt region for the move is the union of the extent of the main window and all attached windows (excluding those with MOVEW in their REJECTMAINCOMS window property).

If an attached window is moved by calling the function MOVEW, it is moved without affecting the main window. If the right-button window menu command Move is called on an attached window, it is passed on to the main window, so that all windows in the group move.

Reshape — If the main window is reshaped, the minimum size of it and all of its attached windows is used as the minimum of the space for the result. Any space greater than the minimum is distributed among the main window and its attached windows. Attached windows with SHAPEW on their REJECTMAINCOMS window property are ignored when finding the minimum size, creating a "ghost" region, or distributing space after a reshape.

If an attached window is reshaped by calling the function SHAPEW, it is reshaped independently. If the right-button window menu command Shape is called on an attached window, it is passed on to the main window, so the whole group is reshaped.

Note: Reshaping the main window will restore the conditions established by the call to ATTACHWINDOW, whereas moving the main window does not. Thus, if A is attached to the top of B and then moved by the user, its new position relative to B will be maintained if B is moved. If B is reshaped, A will be reshaped to the top of B. Additionally, if, while A is moved away from the top of B, C is attached to the top of B, C will position itself above where A used to be.

Close — If the main window is closed, all of the attached windows are closed also and the links from the attached windows to the main window are broken. This is necessary for the windows to be garbage collected.

If an attached window is closed by calling the function CLOSEW, it is closed without affecting the main window. If the right-button window menu command Close is called on an attached window, it is passed on to the main window. Note that closing an attached window detaches it.

Open — If the main window is opened, it opens all attached windows and reestablishes links from them to the main window.

Attached windows can be opened independently and this does not affect the main window. Note that it is possible to reopen a closed attached window and not have it linked to its main window.

Shrink — The collection of windows shrinks as a group. The SHRINKFNs of the attached windows are evaluated but the only icon displayed is the one for the main window.

Redisplay — The main or attached windows can be redisplayed independently.

Totop — If any main or attached window is brought to the top, all of the other windows are brought to the top also.

Expand — Expanding any of the windows expands the whole collection.

Scrolling — All of the windows involved in the group scroll independently.

Clear — All windows clear independently of each other.

Window Properties Of Attached Windows

Windows that are involved in a collection either as a main window or as an attached window have properties stored on them. The only properties that are intended to be set be set by the user are the MINSIZE, MAXSIZE, PASSTOMAINCOMS, and REJECTMAINCOMS window properties. The other properties should be considered read only.

MINSIZE [Window Property]
MAXSIZE [Window Property]

Each of these window properties should be a dotted pair (WIDTH . HEIGHT) or a function to apply to the window that returns a dotted pair. The numbers are used when the main window is reshaped. The MINSIZE is used to determine the size of the smallest region acceptable during reshaping. Any amount greater than the collective minimum is spread evenly among the windows until each reaches MAXSIZE. Any excess is given to the main window.

If you give the main window of an attached window group a MINSIZE or MAXSIZE property, its value is moved to the MAINWINDOWMINSIZE or MAINWINDOWMAXSIZE property, so that the main window can be given a size function that computes the minimum or maximum size of the entire group. Thus, if you want to change the main window's minimum or maximum size after attaching windows to it, you should change the MAINWINDOWMINSIZE or MAINWINDOWMAXSIZE property instead.

This doesn't address the hard problem of overlapping attached windows side to side, for example if window A was attached as [TOP, LEFT] and B as [TOP, RIGHT]. Currently, the attached window functions do not worry about the overlap.

The default MAXSIZE is NIL, which will let the region grow indefinitely.

MAINWINDOW [Window Property]

Pointer from attached windows to the main window of the group. This link is not available if the main window is closed. The function MAINWINDOW is the preferred way to access this property.

ATTACHEDWINDOWS [Window Property]

Pointer from a window to its attached windows. The function ATTACHEDWINDOWS is the preferred way to access this property.

WHEREATTACHED [Window Property]

For attached windows, a dotted pair (EDGE . POSITIONONEDGE) giving the edge and position on the edge that determine how the attached window is placed relative to its main window.

The TOTOPFN window property on attached windows and the properties TOTOPFN, DOSHAPEFN, MOVEFN, CLOSEFN, OPENFN, SHRINKFN, EXPANDFN and CALCULATEREGIONFN on main windows contain functions that implement the attached window manipulation facilities. Care should be used in modifying or replacing these properties.

Communication of Window Menu Commands between Attached Windows is dependent on the name of function used to implement the window command, e.g., CLOSEW implements CLOSE (refer to PASSTOMAINCOMS documentation under Attached Windows). Consequently, if an application intercepts a window command by changing WHENSELECTEDFN for an item in the WindowMenu (for example, to advise the application that a window is being closed), windows may not behave correctly when attached to other windows.

To get around this problem, the Medley release provides the variable `*attached-window-command-synonyms*`. This variable is an alist, where each element is of the form (new-command-function-name . old-command-function-name).

For example, if an application redefines the WindowMenu to call my-close-window when `CLOSE` is selected, that application should:

```
(cl:push '(my-close-window . il:closew) il:*attached-window-
command-synonyms*)
```

in order to tell the attached window system that my-close-window is a synonym function for `CLOSEW`.

28. HARDCOPY FACILITIES

Interlisp-D includes facilities for generating hardcopy in "Interpress" format and "Press" format. Interpress is a file format used for communicating documents to Xerox Network System printers such as the Xerox 8044 and Xerox 5700. Press is a file format used for communicating documents to Xerox laser Xerographic printers known by the names "Dover", "Spruce", "Penguin", and "Raven". There are also library packages available for supporting other types of printer formats (4045, FX-80, C150, etc.). The hardcopy facilities are designed to allow the user to support new types of printers with minimal changes to the user interface.

Files can be in a number of formats, including Interpress files, plain text files, and formatted Tedit files. In order to print a file on a given printer, it is necessary to identify the format of the file, convert the file to a format that the printer can accept, and transmit it. Rather than require that the user explicitly determine file types and do the conversion, the Interlisp-D hardcopy functions generate Interpress or other format output depending on the appropriate choice for the designated printer. The hardcopy functions use the variables PRINTERTYPES and PRINTFILETYPES (described below) to determine the type of a file, how to convert it for a given printer, and how to send it. By changing these variables, the user can define other kinds of printers and print to them using the normal hardcopy functions.

(**SEND.FILE.TO.PRINTER** *FILE HOST PRINTOPTIONS*) [Function]

The function SEND.FILE.TO.PRINTER causes the file *FILE* to be sent to the printer *HOST*. If *HOST* is NIL, the first host in the list DEFAULTPRINTINGHOST which can print *FILE* is used.

PRINTOPTIONS is a property list of the form ($PROP_1$ $VALUE_1$ $PROP_2$ $VALUE_2$ ...). The properties accepted depends on the type of printer. For Interpress printers, the following properties are accepted:

DOCUMENT.NAME The document name to appear on the header page (a string). Default is the full name of the file.

DOCUMENT.CREATION.DATE The creation date to appear on the header page (a Lisp integer date, such as returned by IDATE). The default value is the creation date of the file.

SENDER.NAME The name of the sender to appear on the header page (a string). The default value is the name of the user.

RECIPIENT.NAME The name of the recipient to appear on the header page (a string). The default is none.

MESSAGE An additional message to appear on the header page (a string). The default is none.

#COPIES The number of copies to be printed. The default value is 1.

PAGES.TO.PRINT The pages of the document that should be printed, represented as a list (FIRSTPAGE# LASTPAGE#). For example, if this option is (3 5), this specifies that pages 3 through 5, inclusive, should be printed. Note that the page numbering used for this purpose has no connection to any page numbers that may be printed on the document. The default is to print all of the pages in the document.

MEDIUM The medium on which the master is to be printed. If omitted, this defaults to the value of NSPRINT.DEFAULT.MEDIUM, as follows: NIL means to use the printer's default; T means to use the first medium reported available by the printer; any other value must be a Courier value of type MEDIUM. The format of this type is a list (PAPER (KNOWN.SIZE TYPE)) or (PAPER (OTHER.SIZE (WIDTH LENGTH))). The paper TYPE is one of US.LETTER, US.LEGAL, A0 through A10, ISO.B0 through ISO.B10, and JIS.B0 through JIS.B10. For users who use A4 paper exclusively, it should be sufficient to set NSPRINT.DEFAULT.MEDIUM to (PAPER (KNOWN.SIZE "A4")).

When using different paper sizes, it may be necessary to reset the variable DEFAULTPAGEREGION, the region on the page used for printing (measured in micas from the lower-left corner).

STAPLE? True if the document should be stapled.

#SIDES 1 or 2 to indicate that the document should be printed on one or two sides, respectively. The default is the value of EMPRESS#SIDES.

PRIORITY The priority of this print request, one of LOW, NORMAL, or HIGH. The default is the printer's default.

Note: Press printers only recognize the options #COPIES, #SIDES, DOCUMENT.CREATION.DATE, and DOCUMENT.NAME.

For example,

```
(SEND.FILE.TO.PRINTER 'FOO NIL
   '(#COPIES 3 #SIDES 2 DOCUMENT.NAME "For John"))
```

SEND.FILE.TO.PRINTER calls PRINTERTYPE and PRINTFILETYPE to determine the printer type of *HOST* and the file format of *FILE*. If *FILE* is a formatted file already in a form that the printer can print, it is transmitted directly. Otherwise, CONVERT.FILE.TO.TYPE.FOR.PRINTER is called to do the conversion. [Note: If the file is converted, PRINTOPTIONS is passed to the formatting function, so it can include properties such as HEADING, REGION, and FONTS.] All of these functions use the lists PRINTERTYPES and PRINTFILETYPES to actually determine how to do the conversion.

LISTFILES (Chapter 17) calls the function LISTFILES1 to send a single file to a hardcopy printing device. Interlisp-D is initialized with LISTFILES1 defined to call SEND.FILE.TO.PRINTER.

(**HARDCOPYW** *WINDOW/BITMAP/REGION FILE HOST SCALEFACTOR ROTATION PRINTERTYPE HARDCOPYTITLE*) [Function]

Creates a hardcopy file from a bitmap and optionally sends it to a printer. Note that some printers may have limitations concerning how big or how "complicated" the bitmap may be printed.

WINDOW/BITMAP/REGION can either be a *WINDOW* (open or closed), a *BITMAP*, or a *REGION* (interpreted as a region of the screen). If *WINDOW/BITMAP/REGION* is NIL, the user is prompted for a screen region using GETREGION.

If *FILE* is non-NIL, it is used as the name of the file for output. If *HOST* = NIL, this file is not printed. If *FILE* is NIL, a temporary file is created, and sent to *HOST*.

To save an image on a file without printing it, perform (HARDCOPYW IMAGE FILE). To print an image to the printer PRINTER without saving the file, perform (HARDCOPYW IMAGE NIL PRINTER).

If both *FILE* and *HOST* are NIL, the default action is to print the image, without saving the file. The printer used is determined by the argument PRINTERTYPE and the value of the variable DEFAULTPRINTINGHOST. If PRINTERTYPE is non-NIL, the first host on DEFAULTPRINTINGHOST of the type PRINTERTYPE is used. If PRINTERTYPE is NIL, the first printer on DEFAULTPRINTINGHOST that implements the BITMAPSCALE (as determined by PRINTERTYPES) operation is used, if any. Otherwise, the first printer on DEFAULTPRINTINGHOST is used.

The type of hardcopy file produced is determined by HOST if non-NIL, else by PRINTERTYPE if non-NIL, else by the value of DEFAULTPRINTINGHOST, as described above.

SCALEFACTOR is a reduction factor. If not given, it is computed automatically based on the size of the bitmap and the capabilities of the printer type. This may not be supported for some printers.

ROTATION specifies how the bitmap image should be rotated on the printed page. Most printers (including Interpress printers) only support a ROTATION of multiples of 90.

PRINTERTYPE specifies what type of printer to use when HOST is NIL. HARDCOPYW uses this information to select which printer to use or what print file format to convert the output into, as described above.

The background menu contains a "Hardcopy" command (Chapter 28) that prompts the user for a region on the screen, and sends the image to the default printer.

Hardcopy output may also be obtained by writing a file on the printer device LPT, e.g. (COPYFILE 'FOO '{LPT}). When a file on this device is closed, it is converted to Interpress or some other format (if necessary) and sent to the default printer (the first host on DEFAULTPRINTINGHOST). One can include the printer name directly in the file name, e.g. (COPYFILE 'FOO {LPT}TREMOR:) will send the file to the printer TREMOR:.

HARDCOPYTITLE is a string specifying a title to print on the page containing the screen image. If NIL, the string "Window Image" is used. To omit a title, specify the null string.

(**PRINTERSTATUS** *PRINTER*) [Function]

Returns a list describing the current status of the printer named *PRINTER*. The exact form of the value returned depends on the type of printer. For InterPress printers, the status describes whether the printer is available or busy or needs attention, and what type of paper is loaded in the printer.

Returns NIL if the printer does not respond in a reasonable time, which can occur if the printer is very busy, or does not implement the printer status service.

DEFAULTPRINTINGHOST [Variable]

The variable DEFAULTPRINTINGHOST is used to designate the default printer to be used as the output of printing operations. It should be a list of the known printer host names, for example, (QUAKE LISPPRINT:). If an element of DEFAULTPRINTINGHOST is a list, is interpreted as (PRINTERTYPE HOST), specifying both the host type and the host name. The type of the printer, which determines the protocol used to send to it and the file format it requires, is determined by the function PRINTERTYPE.

If DEFAULTPRINTINGHOST is a single printer name, it is treated as if it were a list of one element.

(**PRINTFILETYPE** *FILE* —) [Function]

Returns the format of the file *FILE*. Possible values include INTERPRESS, TEDIT, etc. If it cannot determine the file type, it returns NIL. Uses the global variable PRINTFILETYPES.

(**PRINTERTYPE** *HOST*) [Function]

Returns the type of the printer *HOST*. Currently uses the following heuristic:

1. If *HOST* is a list, the CAR is assumed to be the printer type and CADR the name of the printer

2. If *HOST* is a litatom with a non-NIL PRINTERTYPE property, the property value is returned as the printer type

3. If *HOST* contains a colon (e.g., PRINTER:PARC:XEROX) it is assumed to be an INTERPRESS printer

4. If *HOST* is the CADR of a list on DEFAULTPRINTINGHOST, the CAR is returned as the printer type

5. Otherwise, the value of DEFAULTPRINTERTYPE is returned as the printer type.

Low-level Hardcopy Variables

The following variables are used to define how Interlisp should generate hardcopy of different types. The user should only need to change these variables when it is necessary to access a new type of printer, or define a new hardcopy document type (not often).

PRINTERTYPES [Variable]

The characteristics of a given printer are determined by the value of the list PRINTERTYPES. Each element is a list of the form

(TYPES (PROPERTY$_1$ VALUE$_1$) (PROPERTY$_2$ VALUE$_2$) ...)

TYPES is a list of the printer types that this entry addresses. The (PROPERTY$_n$ VALUE$_n$) pairs define properties associated with each printer type.

The printer properties include the following:

CANPRINT — Value is a list of the file types that the printer can print directly.

STATUS — Value is a function that knows how to find out the status of the printer, used by PRINTERSTATUS.

PROPERTIES — Value is a function which returns a list of known printer properties.

SEND — Value is a function which invokes the appropriate protocol to send a file to the printer.

BITMAPSCALE — Value is a function of arguments WIDTH and HEIGHT in bits which returns a scale factor for scaling a bitmap.

BITMAPFILE — Value is a form which, when evaluated, converts a bitmap to a file format that the printer will accept.

Note: The name 8044 is defined on PRINTERTYPES as a synonym for the INTERPRESS printer type. The names SPRUCE, PENGUIN, and DOVER are defined on PRINTERTYPES as synonyms for the PRESS printer type. The printer types FULLPRESS and RAVEN are also defined the same as PRESS, except that these printer types indicate that the printer is a "Full Press" printer that is able to scale bitmap images, in addition to the normal Press printer facilities.

PRINTFILETYPES [Variable]

The variable PRINTFILETYPES contains information about various file formats, such as Tedit files and Interpress files. The format is similar to PRINTERTYPES. The properties that can be specified include:

TEST Value is a function which tests a file if it is of the given type. Note that this function is passed an open stream.

CONVERSION Value is a property list of other file types and funcitons that convert from the specified type to the file format.

EXTENSION Value is a list of possible file extensions for files of this type.

29. TERMINALINPUT/OUTPUT

Most input/output operations in Interlisp can be simply modeled as reading or writing on a linear stream of bytes. However, the situation is much more complex when it comes to controlling the user's "terminal," which includes the keyboard, the mouse, and the display screen. For example, Interlisp coordinates the operation of these separate I/O devices so that the cursor on the screen moves as the mouse moves, and any characters typed by the user appear in the window currently containing a flashing cursor. Most of the time, this system works correctly without need for user modification.

The purpose of this chapter is to describe how to access the low-level controls for the terminal I/O devices. It documents the use of interrupt characters, the keyboard characters that generate interrupts. Then, it describes terminal tables, used to determine the meaning of the different editing characters (character delete, line delete, etc.). Then, the "dribble file" facility that allows terminal I/O to be saved onto a file is presented (see the Dribble Files section below). Finally, the low-level functions that control the mouse and cursor, the keyboard, and the screen are documented.

Interrupt Characters

Errors and breaks can be caused by errors within functions, or by explicitly breaking a function. The user can also indicate his desire to go into a break while a program is running by typing certain control characters known as "interrupt characters". The following interrupt characters are currently enabled in Interlisp-D:

Note: In Interlisp-D with multiple processes, it is not sufficient to say that "the computation" is broken, aborted, etc; it is necessary to specify which process is being acted upon. Usually, the user wants interrupts to occur in the TTY process, which is the one currently receiving keyboard input. However, sometimes the user wants to interrupt the mouse process, if it is currently busy executing a menu command or waiting for the user to specify a region on the screen. Most of the interrupt characters below take place in the mouse process if it is busy, otherwise the TTY process. Control-G can be used to break arbitrary processes. For more information, see Chapter 23.

Control-B Causes a break within the mouse process (if busy) or the TTY process. Use Control-G to break a particular process.

Control-D Aborts the mouse process (if busy) or the TTY process, and unwinds its stack to the top level. Calls RESET (see Chapter 14).

Control-E Aborts the mouse process (if busy) or the TTY process, and unwinds its stack to the last ERRORSET. Calls ERROR! (see Chapter 14).

Control-G Pops up a menu listing all of the currently-running processes. Selecting one of the processes will cause a break to take place in that process.

Control-P This interrupt is no longer supported in Medley.

Control-T Flashes the TTY process' window and prints status information for the TTY process. First it prints "**IO wait**," "**Waiting**", or "**Running**," depending on whether the TTY process is currently in waiting for characters to be typed, waiting for some other reason, or running. Next, it prints the names of the top three frames on the stack, to show what is running. Then, it prints a line describing the percentage of time (since the last control-T) that has been spent running a program, swapping, garbage collecting, doing local disk I/O, etc. For example:

Running in TTWAITFORINPUT in TTBIN in TTYIN1

95% Util, 0% Swap, 4% GC

DELETE Clears typeahead in all processes.

The user can disable and/or redefine Interlisp interrupt characters, as well as define new interrupt characters. Interlisp-D is initialized with the following interrupt channels: RESET (**Control-D**), ERROR (**Control-E**), BREAK (**Control-B**), HELP (**Control-G**), PRINTLEVEL (**Control-P**), RUBOUT (**DELETE**), and RAID. Each of these channels independently can be disabled, or have a new interrupt character assigned to it via the function INTERRUPTCHAR described below. In addition, the user can enable new interrupt channels, and associate with each channel an interrupt character and an expression to be evaluated when that character is typed.

(**INTERRUPTCHAR** *CHAR TYP/FORM HARDFLG* —) [Function]

Defines *CHAR* as an interrupt character. If *CHAR* was previously defined as an interrupt character, that interpretation is disabled.

CHAR is either a character or a character code (see Chapter 2). Note that full sixteen-bit NS characters can be specified as interrupt characters (see Chapter 2). *CHAR* can also be a value returned from INTERRUPTCHAR, as described below.

If *TYP/FORM* = NIL, *CHAR* is disabled.

If *TYP/FORM* = T, the current state of *CHAR* is returned without changing or disabling it.

If *TYP/FORM* is one of the literal atoms RESET, ERROR, BREAK, HELP, PRINTLEVEL, RUBOUT, or RAID, then INTERRUPTCHAR assigns *CHAR* to the indicated Interlisp interrupt channel, (reenabling the channel if previously disabled).

If the argument *TYP/FORM* is a symbol designating a predefined system interrupt (RESET, ERROR, BREAK, etc), and HARDFLG is omitted or NIL, then the hardness defaults to the standard hardness of the system interrupt (e.g., MOUSE for the ERROR interrupt).

If *TYP/FORM* is any other literal atom, *CHAR* is enabled as an interrupt character that when typed causes the atom *TYP/FORM* to be immediately set to T.

If *TYP/FORM* is a list, *CHAR* is enabled as a user interrupt character, and *TYP/FORM* is the form that is evaluated when *CHAR* is typed. The interrupt will be hard if *HARDFLG* = T, otherwise soft.

(INTERRUPTCHAR T) restores all Interlisp channels to their original state, and disables all user interrupts.

HARDFLG determines what process the interrupt should run in. If *HARDFLG* is NIL, the interrupt will run in the TTY process, which is the process currently receiving keyboard input. If *HARDFLG* is T, the interrupt will occur in whichever process happens to be running. If *HARDFLG* is MOUSE, the interrupt will happen in the mouse process, if the mouse is busy, otherwise in the TTY process.

INTERRUPTCHAR returns a value which, when given as the *CHAR* argument to INTERRUPTCHAR, will restore things as they were before the call to INTERRUPTCHAR. Therefore, INTERRUPTCHAR can be used in conjunction with RESETFORM or RESETLST (see Chapter 14).

INTERRUPTCHAR is undoable.

(**RESET.INTERRUPTS** *PERMITTEDINTERRUPTS SAVECURRENT?*) [Function]

PERMITTEDINTERRUPTS is a list of interrupt character settings to be performed, each of the form (CHAR TYP/FORM HARDFLG). The effect of RESET.INTERRUPTS is as if (INTERRUPTCHAR CHAR TYP/FORM HARDFLG) were performed for each item on PERMITTEDINTERRUPTS, and (INTERRUPTCHAR OTHERCHAR NIL) were performed on every other existing interrupt character.

If *SAVECURRENT?* is non-NIL, then RESET.INTERRUPTS returns the current state of the interrupts in a form that could be passed to RESET.INTERRUPTS, otherwise it returns NIL. This can be used with a RESET.INTERRUPTS that appears in a RESETFORM, so that the list is built at "entry", but not upon "exit".

(**LISPINTERRUPTS**) [Function]

Returns the initial default interrupt character settings for Interlisp-D, as a list that RESET.INTERRUPTS would accept.

(**INTERRUPTABLE** *FLAG*) [Function]

if *FLAG* = NIL, turns interrupts off. If *FLAG* = T, turns interrupts on. Value is previous setting. INTERRUPTABLE compiles open.

Any interrupt character typed while interrupts are off is treated the same as any other character, i.e., placed in the input buffer, and will not cause an interrupt when interrupts are turned back on.

Terminal Tables

A read table (see Chapter 25) contains input/output information that is media-independent. For example, the action of parentheses is the same regardless of the device from which the input is being performed. A terminal table is an object that contains information that pertains to terminal input/output operations only, such as the character to type to delete the last character or to delete the last line. In addition, terminal tables contain such information as how line-buffering is to be performed, how control characters are to be echoed/printed, whether lowercase input is to be converted to upper case, etc.

Using the functions below, the user may change, reset, or copy terminal tables, or create a new terminal table and install it as the primary terminal table via SETTERMTABLE. However, unlike read tables, terminal tables cannot be passed as arguments to input/output functions.

(**GETTERMTABLE** *TTBL*) [Function]

If *TTBL* = NIL, returns the primary (i.e., current) terminal table. If *TTBL* is a terminal table, return *TTBL*. Otherwise, generates an **ILLEGAL TERMINAL TABLE** error.

(**COPYTERMTABLE** *TTBL*) [Function]

Returns a copy of *TTBL*. *TTBL* can be a real terminal table, NIL (copies the primary terminal table), or ORIG (returns a copy of the original system terminal table). Note that COPYTERMTABLE is the only function that creates a terminal table.

(**SETTERMTABLE** *TTBL*) [Function]

Sets the primary terminal table to be *TTBL*. Returns the previous primary terminal table. Generates an **ILLEGAL TERMINAL TABLE** error if *TTBL* is not a real terminal table.

(**RESETTERMTABLE** *TTBL FROM*) [Function]

Copies (smashes) *FROM* into *TTBL*. *FROM* and *TTBL* can be NIL or a real terminal table. In addition, *FROM* can be ORIG, meaning to use the system's original terminal table.

(**TERMTABLEP** *TTBL*) [Function]

Returns *TTBL*, if *TTBL* is a real terminal table, NIL otherwise.

Terminal Syntax Classes

A terminal table associates with each character a single "terminal syntax class", one of CHARDELETE, LINEDELETE, WORDDELETE, RETYPE, CTRLV, EOL, and NONE. Unlike read table classes, only one character in a particular terminal table can belong to each of the classes (except for the default class NONE). When a new character is assigned one of these syntax classes by SETSYNTAX (see Chapter 25), the previous character is disabled (i.e., reassigned the syntax class NONE), and the value of SETSYNTAX is the code for the previous character of that class, if any, otherwise NIL.

The terminal syntax classes are interpreted as follows:

CHARDELETE (Initially BackSpace and Control-A in Interlisp-D) Typing this character deletes the previous character typed. Repeated use of this character deletes successive characters back to the beginning of the line.

LINEDELETE (Initially Control-Q in Interlisp-D) Typing this character deletes the whole line; it cannot be used repeatedly.

WORDDELETE (Initially Control-W in Interlisp-D) Typing this character deletes the previous "word", i.e., sequence of non-separator characters.

RETYPE (Initially Control-R) Causes the line to be retyped as Interlisp sees it (useful when repeated deletions make it difficult to see what remains).

CTRLV
CNTRLV (Initially Control-V) When followed by A, B, ... Z, inputs the corresponding control character control-A, control-B, ... control-Z. This allows interrupt characters to be input without causing an interrupt.

EOL On input from a terminal, the EOL character signals to the line buffering routine to pass the input back to the calling function. It also is used to terminate inputs to READLINE (see Chapter 13). In general, whenever the phrase carriage-return linefeed is used, what is meant is the character with terminal syntax class EOL.

NONE The terminal syntax class of all other characters.

GETSYNTAX, SETSYNTAX, and SYNTAXP all work on terminal tables as well as read tables (see page X.XX). As with read tables, full sixteen-bit NS characters can be specified in terminal tables (see Chapter 2). When given NIL as a TABLE argument, GETSYNTAX and SYNTAXP use the primary read table or primary terminal table depending on which table contains the indicated CLASS argument. For example, (SETSYNTAX CH 'BREAK) refers to the primary read table, and (SETSYNTAX CH 'CHARDELETE) refers to the primary terminal table. In the absence of such information, all three functions default to the primary read table; e.g., (SETSYNTAX '{ '%[) refers to the primary read table. If given incompatible CLASS and table arguments, all three functions generate errors. For example, (SETSYNTAX CH 'BREAK TTBL), where TTBL is a terminal table, generates an **ILLEGAL READTABLE** error, and (GETSYNTAX 'CHARDELETE RDTBL) generates an **ILLEGAL TERMINAL TABLE** error.

Terminal Control Functions

(**ECHOCHAR** *CHARCODE MODE TTBL*) [Function]

ECHOCHAR sets the "echo mode" of the character *CHARCODE* to *MODE* in the terminal table *TTBL*. The "echo mode" determines how the character is to be echoed or printed. Note that although the name of this function suggests echoing only, it affects all output of the character, both echoing of input and printing of output.

CHARCODE should be a character code. *CHARCODE* can also be a list of characters, in which case ECHOCHAR is applied to each of them with arguments *MODE* and *TTBL*. Note that echo modes can be specified for full sixteen-bit NS characters (see Chapter 2).

MODE should be one of the litatoms IGNORE, REAL, SIMULATE, or INDICATE which specify how the character should be echoed or printed:

IGNORE — *CHARCODE* is never printed.

REAL — *CHARCODE* itself is printed. Some terminals may respond to certain control and meta characters in interesting ways.

SIMULATE — Output of *CHARCODE* is simulated. For example, control-I (tab) may be simulated by printing spaces. The simulation is machine-specific and beyond the control of the user.

INDICATE — For control or meta characters, *CHARCODE* is printed as # and/or ↑ followed by the corresponding alphabetic character. For example, Control-A would echo as ↑A, and meta-Control-W would echo as #↑W.

The value of ECHOCHAR is the previous echo mode for *CHARCODE*. If *MODE* = NIL, ECHOCHAR returns the current echo mode without changing it.

Warning: In some fonts, control and meta characters may be used for printable characters. If the echomode is set to INDICATE for these characters, they will not print out correctly.

(**ECHOCONTROL** *CHAR MODE TTBL*) [Function]

ECHOCONTROL is an old, limited version of ECHOCHAR, that can only specify the echo mode of control characters. *CHAR* is a character or character code. If *CHAR* is an alphabetic character (or code), it refers to the corresponding control character, e.g., (ECHOCONTROL 'Z 'INDICATE) if equivalent to (ECHOCHAR (CHARCODE ↑Z) 'INDICATE).

(**ECHOMODE** *FLG TTBL*) [Function]

If *FLG* = T, turns echoing for terminal table *TTBL* on. If *FLG* = NIL, turns echoing off. Returns the previous setting.

Note: Unlike ECHOCHAR, this only affects echoing of typed-in characters, not printing of characters.

(**GETECHOMODE** *TTBL*) [Function]

Returns the current echo mode for *TTBL*.

The following functions manipulate the "raise mode," which determines whether lower case characters are converted to upper case when input from the terminal. There is no "raise mode" for input from files.

(**RAISE** *FLG TTBL*) [Function]

Sets the RAISE mode for terminal table *TTBL*. If *FLG* = NIL, all characters are passed as typed. If *FLG* = T, input is echoed as typed, but lowercase letters are converted to upper case. If *FLG* = 0, input is converted to uppercase before it is echoed. Returns the previous setting.

(**GETRAISE** *TTBL*) [Function]

Returns the current RAISE mode for *TTBL*.

(**DELETECONTROL** *TYPE MESSAGE TTBL*) [Function]

Specifies the output protocol when a CHARDELETE or LINEDELETE is typed, by specifying character strings to print when characters are deleted.

Interlisp-10 (designed for use on hardcopy terminals) echos the characters being deleted, preceding the first by a \ and following the last by a \, so that it is easy to see exactly what was deleted. Interlisp-D is initially set up to physically erase the deleted characters from the display, so the DELETECONTROL strings are initialized to the null string.

The various values of *TYPE* specify different phases of the deletion, as follows:

1STCHDEL — *MESSAGE* is the message printed the first time CHARDELETE is typed. Initially "\" in Interlisp-10.

NTHCHDEL — *MESSAGE* is the message printed when the second and subsequent CHARDELETE characters are typed (without intervening characters). Initially "" in Interlisp-10.

POSTCHDEL — *MESSAGE* is the message printed when input is resumed following a sequence of one or more CHARDELETE characters. Initially "\" in Interlisp-10.

EMPTYCHDEL — *MESSAGE* is the message printed when a CHARDELETE is typed and there are no characters in the buffer. Initially "## cr" in Interlisp-10.

ECHO — If *TYPE* = ECHO, the characters deleted by CHARDELETE are echoed. *MESSAGE* is ignored.

NOECHO — If *TYPE* = NOECHO, the characters deleted by CHARDELETE are not echoed. *MESSAGE* is ignored.

LINEDELETE — *MESSAGE* is the message printed when the LINEDELETE character is typed. Initially "## cr".

Note: In Interlisp-10, the LINEDELETE, 1STCHDEL, NTHCHDEL, POSTCHDEL, and EMPTYCHDEL messages must be 4 characters or fewer in length.

DELETECONTROL returns the previous message as a string. If *MESSAGE* = NIL, the value returned is the previous message without changing it. For *TYPE* = ECHO and NOECHO, the value of DELETECONTROL is the previous echo mode, i.e., ECHO or NOECHO.

(**GETDELETECONTROL** *TYPE TTBL*) [Function]

Returns the current DELETECONTROL mode for *TYPE* in *TTBL*.

Line-Buffering

Characters typed at the terminal are stored in two buffers before they are passed to an input function. All characters typed in are put into the low-level "system buffer", which allows type-ahead. When an input function is entered, characters are transferred to the "line buffer" until a character with terminal syntax class EOL appears (or, for calls from READ, when the count of unbalanced open parentheses reaches 0). Note that PEEKC is an exception; it returns the character immediately when its second argument is NIL. Until this time, the user can delete characters one at a time from the line buffer by typing the current CHARDELETE character, or delete the entire line buffer back to the last carriage-return by typing the current LINEDELETE.

This line editing is not performed by READ or RATOM, but by Interlisp, i.e., it does not matter (nor is it necessarily known) which function will ultimately process the characters, only that they are still in the Interlisp line buffer. However, the function that is requesting input at the time the buffering starts does determine whether parentheses counting is observed. For example, if a program performs (PROGN (RATOM) (READ)) and the user types in "A (B C D)", the user must type in the carriage-return following the right parenthesis before any action is taken, because the line buffering is happening under RATOM. If the program had performed (PROGN (READ) (READ)), the line-buffering would be under READ, so that the right parenthesis would terminate line buffering, and no terminating carriage-return would be required.

Once a carriage-return has been typed, the entire line is "available" even if not all of it is processed by the function initiating the request for input. If any characters are "left over", they are returned immediately on the next request for input. For example, (LIST (RATOM) (READC) (RATOM)) when the input is "A Bcr" returns the three-element list (A % B) and leaves the carriage-return in the buffer.

If a carriage-return is typed when the input under READ is not "complete" (the parentheses are not balanced or a string is in progress), line buffering continues, but the lines completed so far are not available for editing with CHARDELETE or LINEDELETE.

The function CONTROL is available to defeat line-buffering:

(**CONTROL** *MODE TTBL*) [Function]

If *MODE* = T, eliminates Interlisp's normal line-buffering for the terminal table *TTBL*. If *MODE* = NIL, restores line-buffering (normal). When operating with a terminal table in which (CONTROL T) has been performed, characters are returned to the calling function without line-buffering as described below.

CONTROL returns its previous setting.

(**GETCONTROL** *TTBL*) [Function]

Returns the current control mode for *TTBL*.

The function that initiates the request for input determines how the line is treated when (CONTROL T) is in effect:

READ If the expression being typed is a list, the effect is the same as though done with (CONTROL NIL), i.e., line-buffering continues until a carriage-return or matching parentheses. If the expression being typed is not a list, it is returned as soon as a break or separator character is encountered, e.g., (READ) when the input is "ABC<space>" immediately returns ABC. CHARDELETE and LINEDELETE are available on those characters still in the buffer. Thus, if a program is performing several reads under (CONTROL T), and the user types "NOW IS THE TIME" followed by Control-Q, only TIME is deleted, since the rest of the line has already been transmitted to READ and processed.

An exception to the above occurs when the break or separator character is an opening parenthesis, bracket or double-quote, since returning at this point would leave the line buffer in a "funny" state. Thus if the input to (READ) is "ABC(", the ABC is not read until a carriage-return or matching parentheses is encountered. In this case the user could LINEDELETE the entire line, since all of the characters are still in the buffer.

RATOM Characters are returned as soon as a break or separator character is encountered. Until then, LINEDELETE and CHARDELETE may be used as with READ. For example, (RATOM) followed by "ABC<control-A><space>" returns AB. (RATOM) followed by "(<control-A>" returns (and types ## indicating that control-A was attempted with nothing in the buffer, since the (is a break character and would therefore already have been read.

READC
PEEKC The character is returned immediately; no line editing is possible. In particular, (READC) is perfectly happy to return

the CHARDELETE or LINEDELETE characters, or the ESCAPE character (%).

The system buffer and line buffer can be directly manipulated using the following functions.

(**CLEARBUF** *FILE FLG*) [Function]

Clears the input buffer for *FILE*. If *FILE* is T and *FLG* is T, the contents of Interlisp's system buffer and line buffer are saved (and can be obtained via SYSBUF and LINBUF described below).

When you type Control-D or Control-E, or any of the interrupt characters that require terminal interaction (Control-G, or Control-P), Interlisp automatically performs (CLEARBUF T T). For Control-P and, when the break is exited normally, control-H, Interlisp restores the buffer after the interaction.

The action of (CLEARBUF T), i.e., clearing of typeahead, is also available as the RUBOUT interrupt character, initially assigned to the delete key in Interlisp-D. Note that this interrupt clears both buffers at the time it is typed, whereas the action of the CHARDELETE and LINEDELETE character occur at the time they are read.

(**SYSBUF** *FLG*) [Function]

If *FLG* = T, returns the contents of the system buffer (as a string) that was saved at the last (CLEARBUF T T). If *FLG* = NIL, clears this internal buffer.

(**LINBUF** *FLG*) [Function]

Same as SYSBUF for the line buffer.

If both the system buffer and Interlisp's line buffer are empty, the internal buffers associated with LINBUF and SYSBUF are not changed by a (CLEARBUF T T).

(**BKSYSBUF** *X FLG RDTBL*) [Function]

BKSYSBUF appends the PRIN1-name of *X* to the system buffer. The effect is the same as though the user typed *X*. Some implementations have a limit on the length of *X*, in which case characters in *X* beyond the limit are ignored. Returns *X*.

If *FLG* is T, then the PRIN2-name of *X* is used, computed with respect to the readtable *RDTBL*. If *RDTBL* is NIL or omitted, the current readtable of the TTY process (which is to receive the characters) is used. Use this for copy selection functions that want their output to be a readable expression in an Exec.

Note that if you are typing at the same time as the BKSYSBUF is being performed, the relative order of the typein and the characters of *X* is unpredictable.

(**BKLINBUF** *STR*) [Function]

STR is a string. BKLINBUF sets Interlisp's line buffer to *STR*. Some implementations have a limit on the length of *STR*, in which case characters in *STR* beyond the limit are ignored. Returns *STR*.

(**BKSYSCHARCODE** *CODE*) [Function]

This function appends the character code *CODE* to the system input buffer. The function BKSYSBUF is implemented by repeated calls to BKSYSCHARCODE.

BKLINBUF, BKSYSBUF, LINBUF, and SYSBUF provide a way of "undoing" a CLEARBUF. Thus to "peek" at various characters in the buffer, one could perform (CLEARBUF T T), examine the buffers via LINBUF and SYSBUF, and then put them back.

The more common use of these functions is in saving and restoring typeahead when a program requires some unanticipated (from the user's standpoint) input. The function RESETBUFS provides a convenient way of simply clearing the input buffer, performing an interaction with the user, and then restoring the input buffer.

(**RESETBUFS** *$FORM_1$, $FORM_2$,... FORMN*) [NLambda NoSpread Function]

Clears any typeahead (ringing the terminal's bell if there was, indeed, typeahead), evaluates *$FORM_1$*, *$FORM_2$*, ... *$FORM_N$*, then restores the typeahead. Returns the value of *$FORM_N$*. Compiles open.

Dribble Files

A dribble file is a "transcript" of all of the input and output on a terminal. In Interlisp-D, DRIBBLE opens a dribble file for the current process, recording the terminal input and output for that process. Multiple processes can have separate dribble files open at the same time.

(**DRIBBLE** *FILE APPENDFLG THAWEDFLG*) [Function]

Opens *FILE* and begins recording the typescript. Returns the old dribble file if any, otherwise NIL. If *APPENDFLG* = T, the typescript will be appended to the end of *FILE*. If *THAWEDFLG* = T, the file will be opened in "thawed" mode, for those implementations that support it. (DRIBBLE) closes the dribble file for the current process. Only one dribble file can be active for each process at any one time, so (DRIBBLE FILE1) followed by (DRIBBLE FILE2) will cause FILE1 to be closed.

(**DRIBBLEFILE**) [Function]

Returns the name of the current dribble file for the current process, if any, otherwise NIL.

Terminal input is echoed to the dribble file a line buffer at a time. Thus, the typescript produced is somewhat neater than that appearing on the user's terminal, because it does not show characters that were erased via Control-A or Control-Q. Note that the typescript file is not included in the list of files returned by (OPENP), nor will it be closed by a call to CLOSEALL or CLOSEF. Only (DRIBBLE) closes the typescript file.

Cursor and Mouse

A mouse is a small box connected to the computer keyboard by a long wire. On the top of the mouse are two or three buttons. On the bottom is a rolling ball or a set of photoreceptors, to detect when the mouse is moved. As the mouse is moved on a surface, a small image on the screen, called the cursor, moves to follow the movement of the mouse. By moving the mouse, the user can cause the cursor to point to any part of the display screen.

The mouse and cursor are an important part of the Interlisp-D user interface. The Interlisp-D window system allows the user to create, move, and reshape windows, and to select items from displayed menus, all by moving the mouse and clicking the mouse buttons. This section describes the low-level functions used to control the mouse and cursor.

Changing the Cursor Image

Interlisp-D maintains the image of the cursor on the screen, moving it as the mouse is moved. The bitmap that becomes visible as the cursor can be accessed by the following function:

(**CURSORBITMAP**) [Function]

Returns the cursor bitmap.

CURSORWIDTH [Variable]
CURSORHEIGHT [Variable]

Value is the width and height of the cursor bitmap, respectively.

The cursor bitmap can be changed like any other bitmap by BITBLTing into it or pointing a display stream at it and printing or drawing curves. The CURSOR datatype has the following field names CUBITSPERPIXEL CUIMAGE, CUMASK, CUHOTSPOTX, CUHOTSPOTY, CUDATA

CURSOR objects can be saved on a file using the file package command CURSORS, or the UGLYVARS file package command.

(**CURSORCREATE** *BITMAP X Y*) [Function]

Returns a cursor object which has *BITMAP* as its image and the location (*X, Y*) as the hot spot. If *X* is a POSITION, it is used as the hot spot. If *BITMAP* has dimensions different from CURSORWIDTH by CURSORHEIGHT, the lesser of the widths and the lesser of the heights are used to determine the bits that actually get copied into the lower left corner of the cursor. If *X* is NIL, 0 is used. If *Y* is NIL, CURSORHEIGHT-1 is used. The default cursor is an uparrow with its tip in the upper left corner and its hot spot at (0,CURSORHEIGHT-1).

(**CURSOR** *NEWCURSOR* —) [Function]

Returns a CURSOR record instance that contains (a copy of) the current cursor specification. If *NEWCURSOR* is a CURSOR record instance, the cursor will be set to the values in *NEWCURSOR*. If *NEWCURSOR* is T, the cursor will be set to the default cursor DEFAULTCURSOR, an upward left pointing arrow: .

(**SETCURSOR** *NEWCURSOR* —) [Function]

If *NEWCURSOR* is a CURSOR record instance, the cursor will be set to the values in *NEWCURSOR*. This does not return the old cursor, and therefore, provides a way of changing the cursor without using storage.

(**FLIPCURSOR**) [Function]

Inverts the cursor.

The following list describes the cursors used by the Interlisp-D system. Most of them are stored as the values of various variables.

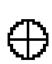
In variable DEFAULTCURSOR. This is the default cursor.

In variable WAITINGCURSOR. Represents an hourglass. Used during long computations.

In variable MOUSECONFIRMCURSOR. Indicates that the system is waiting for the user to confirm an action by pressing the left mouse button, or aborting the action by pressing any other button. Used by the function MOUSECONFIRM (see Chapter 28).

In variable SYSOUTCURSOR. Indicates that the system is saving the virtual memory in a sysout file. See SYSOUT, Chapter 12.

In variable SAVINGCURSOR. Indicates that SAVEVM has been called automatically to save the virtual memory state after the system is idle for long enough. See SAVEVMWAIT, Chapter 12.



In variable CROSSHAIRS. Used by GETPOSITION (see Chapter 28) to indicate a position.

In variable BOXCURSOR. Used by GETBOXPOSITION (see Chapter 28) to indicate where to place the corner of a box.

In variable FORCEPS. Used by GETREGION (see Chapter 28) when the user switches corners.

In variable EXPANDINGBOX. Used by GETREGION (see Chapter 28) when a box is first displayed.

In variable UpperRightCursor.

In variable LowerRightCursor.

In variable UpperLeftCursor.

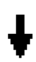
In variable LowerLeftCursor.

The previous four cursors are used by GETREGION (see Chapter 28) to indicate the four corners of a region.

In variable VertThumbCursor. Used during scrolling to indicate thumbing in a vertical scroll bar.

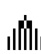
In variable VertScrollCursor.

In variable ScrollUpCursor.



In variable ScrollDownCursor.

The previous four cursors are used by SCROLL.HANDLER (see Chapter 28) during vertical scrolling.



In variable HorizThumbCursor. Used during scrolling to indicate thumbing in a horizontal scroll bar.

In variable HorizScrollCursor.

In variable ScrollLeftCursor.

In variable ScrollRightCursor.

The previous four cursors are used by SCROLL.HANDLER (see Chapter 28) during horizontal scrolling.

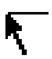
TELE
RAID, ↑D, ↑N, CMD

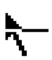
CA
Raid, Brk, N'T These cursors are used by the Teleraid low-level debugger. These cursors are not accessable as standard Interlisp-D cursors.

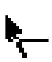
Flashing Bars on the Cursor

The low-level Interlisp-D system uses the cursor to display certain system status information, such as garbage collection or swapping. This is done because changing the cursor image is very quick, and does not require interacting with the window system. Interlisp inverts horizontal bars on the cursor when the system is swapping pages, or doing certain stack operations. Normally, these bars are only inverted for a very short time, so they look like they are flashing. These cursor changes are interpreted as follows:

Inverted cursor:  Whatever image is being displayed as the cursor, whenever Interlisp does a garbage collection, the whole cursor is inverted.

Top bar:  Swap read. On when Interlisp is swapping in a page from the virtual memory file into the real memory. It is also on when Interlisp allocates a new virtual memory page, even though that doesn't involve a disk read. If this is flashing a lot, the system is doing a lot of swapping. This is an indication that the virtual memory working set is fragmented (see Chapter 22). Performance may be improved by reloading a clean Interlisp system.

Upper middle bar:  Stack operations. If this is flashing a lot, it suggests that some process is neglecting to release stack pointers in a timely fashion (see Chapter 11).

Lowereler middle bar:  Stack operations. On when Interlisp is moving frames on the stack. If the system is slow, and this is flashing a lot, HARDRESET (see Chapter 23) sometimes helps.

Bottom bar:  Swap write. On when Interlisp writes a dirty virtual memory page from the real memory back into the virtual memory file.

Cursor Position

The position at which the cursor bitmap is being displayed can be read or set using the following functions:

(**CURSORPOSITION** *NEWPOSITION DISPLAYSTREAM OLDPOSITION*) [Function]

Returns the location of the cursor in the coordinate system of *DISPLAYSTREAM* (or the current display stream, if *DISPLAYSTREAM* is NIL). If *NEWPOSITION* is non-NIL, it should be a position and the cursor will be positioned at *NEWPOSITION*. If *NEWPOSITION* is NIL, the current position is simple returned.

The current position of the cursor is the position of the "hot spot" of the cursor, not the position of the cursor bitmap.

If *OLDPOSITION* is a POSITION object, this object will be changed to point to the location of the cursor and returned, rather of allocating a new POSITION. This can improve performance if CURSORPOSITION is called repeatedly to track the cursor.

To get the location of the cursor in absolute screen coordinates, use the variables LASTMOUSEX and LASTMOUSEY.

(**ADJUSTCURSORPOSITION** *DELTAX DELTAY*) [Function]

Moves the cursor *DELTAX* points in the X direction and *DELTAY* points in the Y direction. *DELTAX* and *DELTAY* default to 0.

Mouse Button Testing

There are two or three keys on the mouse. These keys (also called buttons) are referred to by their location: LEFT, MIDDLE, or RIGHT. The following macros are provided to test the state of the mouse buttons:

(**MOUSESTATE** *BUTTONFORM*) [Macro]

Reads the state of the mouse buttons, and returns T if that state is described by *BUTTONFORM*. *BUTTONFORM* can be one of the key indicators LEFT, MIDDLE, or RIGHT; the atom UP (indicating all keys are up); the form (ONLY KEY); or a form of AND, OR, or NOT applied to any valid button form.

For example: (MOUSESTATE LEFT) will be true if the left mouse button is down. (MOUSESTATE (ONLY LEFT)) will be true if the left mouse button is the only one down. (MOUSESTATE (OR (NOT LEFT) MIDDLE)) will be true if either the left mouse button is up or the middle mouse button is down.

(**LASTMOUSESTATE** *BUTTONFORM*) [Macro]

Similar to MOUSESTATE, but tests the value of LASTMOUSEBUTTONS (below) rather than getting the current state. This is useful for determining which keys caused MOUSESTATE to be true.

(**UNTILMOUSESTATE** *BUTTONFORM INTERVAL*) [Macro]

BUTTONFORM is as described in MOUSESTATE. Waits until *BUTTONFORM* is true or until *INTERVAL* milliseconds have elapsed. The value of UNTILMOUSESTATE is T if *BUTTONFORM* was satisfied before it timed out, otherwise NIL. If *INTERVAL* is NIL, it waits indefinitely. This compiles into an open loop that calls the TTY wait background function. This form should not be used inside the TTY wait background function. UNTILMOUSESTATE does not use any storage during its wait loop.

Low Level Mouse Functions

This section describes the functions and variables that provide low level access to the mouse and cursor.

(**LASTMOUSEX** *DISPLAYSTREAM*) [Function]

Returns the value of the cursor's X position in the coordinates of *DISPLAYSTREAM* (as of the last call to GETMOUSESTATE, below).

(**LASTMOUSEY** *DISPLAYSTREAM*) [Function]

Returns the value of the cursor's Y position in the coordinates of *DISPLAYSTREAM* (as of the last call to GETMOUSESTATE, below).

LASTMOUSEX [Variable]

Value is the X position of the cursor in absolute screen coordinates (as of the last call to GETMOUSESTATE, below).

LASTMOUSEY [Variable]

Value is the Y position of the cursor in absolute screen coordinates (as of the last call to GETMOUSESTATE, below).

LASTMOUSEBUTTONS [Variable]

Value is an integer that has bits on corresponding to the mouse buttons that are down (as of the last call to GETMOUSESTATE, below). Bit 4Q is the left mouse button, 2Q is the right button, 1Q is the middle button.

LASTKEYBOARD [Variable]

Value is an integer encoding the state of certain keys on the keyboard (as of the last call to GETMOUSESTATE, below). Bit 200Q = lock, 100Q = left shift, 40Q = ctrl, 10Q = right shift, 4Q = blank Bottom, 2Q = blank Middle, 1Q = blank Top. If the key is down, the corresponding bit is on.

(**GETMOUSESTATE**) [Function]

Reads the current state of the mouse and sets the variables LASTMOUSEX, LASTMOUSEY, and LASTMOUSEBUTTONS. In polling mode, the program must remember the previous state and look for changes, such as a key going up or down, or the cursor moving outside a region of interest.

(**DECODEBUTTONS** *BUTTONSTATE*) [Function]

Returns a list of the mouse buttons that are down in the state *BUTTONSTATE*. If *BUTTONSTATE* is not a small integer, the value of `LASTMOUSEBUTTONS` (above) is used. The button names that can be returned are: `LEFT`, `MIDDLE`, `RIGHT` (the three mouse keys).

Keyboard Interpretation

For each key on the keyboard and mouse there is a corresponding bit in memory that the hardware turns on and off as the key moves up and down. System-level routines decode the meaning of key transitions according to a table of "key actions", which may be to put particular character codes in the sysbuffer, cause interrupts, change the internal shift / control status, or create events to be placed in the mouse buffer.

(**KEYDOWNP** *KEYNAME*) [Function]

Used to read the instantaneous state of any key, independent of any buffering or pre-assigned key action. Returns `T` if the key named *KEYNAME* is down at the moment the function is executed.

Most keys are named by the characters on the key-top. Therefore, `(KEYDOWNP 'a)` or `(KEYDOWNP 'A)` returns `T` if the "A" key is down.

There are a number of keys that do not have standard names printed on them. These can be accessed by special names as follows:

Space	`SPACE`
Carriage return	`CR`
Line-feed	`LF`
Backspace	`BS`
Tab	`TAB`
Blank keys on 1132	The 1132 keyboard has three unmarked keys on the right of the normal keyboard. These can be accessed by `BLANK-BOTTOM`, `BLANK-MIDDLE`, and `BLANK-TOP`.
Escape	`ESCAPE`
Shift keys	`LSHIFT` for the left shift key, `RSHIFT` for the right shift key.
Shift lock key	`LOCK`
Control key	`CTRL`
Mouse buttons	The state of the mouse buttons can be accessed using `LEFT`, `MIDDLE`, and `RIGHT`.

If KEYNAME is a small integer, it is taken to be the internal key number. Otherwise, it is taken to be the name of the key. This means, for example, that the name of the "6" key is not the number 6. Instead, spelled-out names for all the digit keys have been assigned. The "6" key is named SIX. It happens that the key number of the "6" key is 2. Therefore, the following two forms are equivalent:

```
(KEYDOWNP 'SIX)

(KEYDOWNP 2)
```

(**SHIFTDOWNP** *SHIFT*) [Function]

Returns T if the internal "shift" flag specified by *SHIFT* is on; NIL otherwise.

If *SHIFT* = 1SHIFT, 2SHIFT, LOCK, META, or CTRL, SHIFTDOWNP returns the state of the left shift, right shift, shift lock, control, and meta flags, respectively.

If *SHIFT* = *SHIFT*, SHIFTDOWNP returns T if either the left or right shift flag is on.

If *SHIFT* = USERMODE1, USERMODE2, or USERMODE3, SHIFTDOWNP returns the state of one of three user-settable flags that have no other effect on key interpretation. These flags can be set or cleared on character transitions by using KEYACTION (below).

(**KEYACTION** *KEYNAME ACTIONS* —) [Function]

Changes the internal tables that define the action to be taken when a key transition is detected by the system keyboard handler. *KEYNAME* is specified as for KEYDOWNP. *ACTIONS* is a dotted pair of the form (DOWN-ACTION . UP-ACTION), where the acceptable transition actions and their interpretations are:

NIL

IGNORE Take no action on this transition (the default for up-transitions on all ordinary characters).

(CHAR SHIFTEDCHAR LOCKFLAG)

If a transition action is a three-element list, CHAR and SHIFTEDCHAR are either character codes or (non-numeric) single-character litatoms standing for their character codes. Note that CHAR and SHIFTEDCHAR can be full sixteen-bit NS characters (see page X.XX). When the transition occurs, CHAR or SHIFTEDCHAR is transmitted to the system buffer, depending on whether either of the two shift keys are down.

LOCKFLAG is optional, and may be LOCKSHIFT or NOLOCKSHIFT. If LOCKFLAG is LOCKSHIFT, then SHIFTEDCHAR will also be transmitted when the LOCK shift is down (the alphabetic keys initially specify LOCKSHIFT, but the digit keys specify NOLOCKSHIFT). For example, (a A LOCKSHIFT) and (61Q ! NOLOCKSHIFT) are the initial settings for the down transitions of the "a" and "1" keys respectively.

1SHIFTUP, 1SHIFTDOWN

2SHIFTUP, 2SHIFTDOWN

CTRLUP, CTRLDOWN

METAUP, METADOWN Change the status of the internal "shift" flags for the left shift, right shift, control, and meta keys, respectively. These shifts affect the interpretation of ordinary key actions. If either of the shifts is down, then SHIFTEDCHARs are transmitted. If the control flag is on, then the the seventh bit of the character code is cleared as characters are transmitted. If the meta flag is on, the the eighth bit of the character code is set (normally cleared) as characters are transmitted. For example, the initial keyactions for the left shift key is (1SHIFTDOWN . 1SHIFTUP).

LOCKUP, LOCKDOWN, LOCKTOGGLE

Change the status of the internal "shift" flags for the shift lock key. If the lock flag is down, then SHIFTEDCHARs are transmitted if the key action specified LOCKSHIFT. LOCKUP and LOCKDOWN clear and set the shift lock flag, respectively. LOCKTOGGLE complements the flag (turning it off if the flag is on; on if the flag is off).

USERMODE1UP, USERMODE1DOWN, USERMODE1TOGGLE

USERMODE2UP, USERMODE2DOWN, USERMODE2TOGGLE

USERMODE3UP, USERMODE3DOWN, USERMODE3TOGGLE

Change the status of the three user flags USERMODE1, USERMODE2, and USERMODE3, whose status can be determined by calling SHIFTDOWNP (above). These flags have no other effect on key interpretation.

EVENT An encoding of the current state of the mouse and selected keys is placed in the mouse-event buffer when this transition is detected.

KEYACTION returns the previous setting for KEYNAME. If ACTIONS is NIL, returns the previous setting without changing the tables.

(**MODIFY.KEYACTIONS** *KEYACTIONS SAVECURRENT?*) [Function]

KEYACTIONS is a list of key actions to be set, each of the form (KEYNAME . ACTIONS). The effect of MODIFY.KEYACTIONS is as if (KEYACTION KEYNAME ACTIONS) were performed for each item on *KEYACTIONS*.

If *SAVECURRENT?* is non-NIL, then MODIFY.KEYACTIONS returns a list of all the results from *KEYACTION*, otherwise it returns NIL. This can be used with a MODIFY.KEYACTIONS that appears in a RESETFORM, so that the list is built at "entry", but not upon "exit".

(**METASHIFT** *FLG*) [NoSpread Function]

If *FLG* is T, changes the keyboard handler (via KEYACTION) so as to interpret the "stop" key on the 1108 as a metashift: if a key is struck while the meta is down, it is read with the 200Q bit set. For CHAT users this is a way of getting an "Edit" key on your simulated Datamedia.

If *FLG* is other than NIL or T, it is passed as the ACTIONS argument to KEYACTION. The reason for this is that if someone has set the "STOP" key to some random behavior, then (RESETFORM (METASHIFT T) --) will correctly restore that random behavior.

Display Screen

Medley supports a high-resolution bitmap display screen. All printing and drawing operations to the screen are actually performed on a bitmap in memory, which is read by the computer hardware to become visible as the screen. This section describes the functions used to control the appearance of the display screen.

(**SCREENBITMAP**) [Function]

Returns the screen bitmap.

SCREENWIDTH [Variable]
SCREENHEIGHT [Variable]

Value is the width and height of the screen bitmap, respectively.

WHOLEDISPLAY [Variable]

Value is a region that is the size of the screen bitmap.

The background shade of the display window can be changed using the following function:

(**CHANGEBACKGROUND** *SHADE* —) [Function]

Changes the background shade of the window system. *SHADE* determines the pattern of the background. If *SHADE* is a texture, then the background is simply painted with it. If *SHADE* is a BITMAP, the background is tesselated (tiled) with it to cover the screen. If *SHADE* is T, it changes to the original shade, the value of WINDOWBACKGROUNDSHADE. It returns the previous value of the background.

WINDOWBACKGROUNDSHADE [Variable]

Value is the default background shade for the display.

(**VIDEOCOLOR** *BLACKFLG*) [NoSpread Function]

Sets the interpretation of the bits in the screen bitmap. If *BLACKFLG* is NIL, a 0 bit will be displayed as white, otherwise a 0 bit will be displayed as black. VIDEOCOLOR returns the previous setting. If *BLACKFLG* is not given, VIDEOCOLOR will return the current setting without changing anything.

Note: This function only works on the Xerox 1100 and Xerox 1108.

(**VIDEORATE** *TYPE*) [Function]

Sets the rate at which the screen is refreshed. *TYPE* is one of NORMAL or TAPE. If *TYPE* is TAPE, the screen will be refreshed at the same rate as TV (60 cycles per second). This makes the picture look better when video taping the screen. Note: Changing the rate may change the dimensions of the display on the picture tube.

Maintaining the video image on the screen uses cpu cycles, so turning off the display can improve the speed of compute-bound tasks. When the display is off, the screen will be white but any printing or displaying that the program does will be visible when the display is turned back on.

Note: Breaks and PAGEFULLFN waiting (see Chapter 28) turn the display on, but users should be aware that it is possible to have the system waiting for a response to a question printed or a menu displayed on a non-visible part of the screen. The functions below are provided to turn the display off.

Note: These functions have no effect on the Xerox 1108 display.

(**SETDISPLAYHEIGHT** *NSCANLINES*) [Function]

Sets the display to only show the top *NSCANLINES* of the screen. If *NSCANLINES* is T, resets the display to show the full screen. Returns the previous setting.

(**DISPLAYDOWN** *FORM NSCANLINES*) [Function]

Evaluates *FORM* (with the display set to only show the top *NSCANLINES* of the screen), and returns the value of *FORM*. It restores the screen to its previous setting. If *NSCANLINES* is not given, it defaults to 0.

Miscellaneous Terminal I/O

(**RINGBELLS** *N*) [Function]

Flashes (reverse-videos) the screen *N* times (default 1). On the Xerox 1108, this also beeps through the keyboard speaker.

(**PLAYTUNE** *Frequency/Duration.pairlist*) [Function]

On the Xerox 1108, PLAYTUNE plays a sequence of notes through the keyboard speaker. *Frequency/Duration.pairlist* should be a list of dotted pairs (*FREQUENCY . DURATION*). PLAYTUNE maps down its argument, beeping the 1108 keyboard buzzer at each frequency for the specified amount of time. Specifying NIL for a frequency means to turn the beeper off the specified amount of time. The units of time are TICKS (Chapter 12), which last about 28.78 microseconds on the Xerox 1108. PLAYTUNE makes no sound on a Xerox 1132. The default "simulate" entry for Control-G (ASCII BEL) on the 1108 uses PLAYTUNE to make a short beep.

PLAYTUNE is implemented using BEEPON and BEEPOFF:

(**BEEPON** *FREQ*) [Function]

On the Xerox 1108, turns on the keyboard speaker playing a note with frequency *FREQ*, measured in Hertz (see Chapter 12). The speaker will continue to play the note until BEEPOFF is called.

(**BEEPOFF**) [Function]

Turns off the keyboard speaker on the Xerox 1108.

(**SETMAINTPANEL** *N*) [Function]

On the Xerox 1108, this sets the four-digit "maintanance panel" display on the front of the computer to display the number *N*.

Inspecting and Repairing Notefiles

Xerox Corporation

Randy Trigg

[First written: 8/9/85 Randy Trigg]
[Last updated: 8/26/85 Randy Trigg]

This document describes the Notefile inspector facility available via the Inspect&Repair option on the Notefile Ops menu in NoteCards Release1.2i.

The old Repair Notefile facility rebuilt the links in a Notefile from the contents of card substances. This was used whenever a notefile was thought to have inconsistent links. The problem was that notefiles with inconsistent links often had other problems that caused Repair to break. Thus the motivation for developing the notefile inspector documented here was to verify a notefile's readability before invoking the link rebuilder. As it turns out, this inspector is useful generally for checking the health of a notefile, deleting cards and backing up other cards (or more precisely, card parts) to previous versions. Thus, you may want to use the inspector even if your notefile is healthy and doesn't need its links rebuilt.

The notefile inspector has three separate phases: reading the notefile's data area searching for healthy card parts, allowing the user to make modifications, and rebuilding the links. The process can be aborted after phase 1 or 2 if desired. This document begins with a brief discussion of the organization of a notefile. Then follows sections describing each of the three phases. Finally, I outline some tips, strategies and pitfalls to watch for.

1. What you need to know about a notefile's innards.

1.1 Notefile structure.

A notefile consists of two parts, the index and the data area. Each card in the notefile has an entry in the index. An index entry has 5 parts, a status field and 4 pointers. The status field specifies whether the index entry is free or occupied by an active or deleted card. There is one pointer in the index entry to each of the 4 parts of a card: substance, title, links and property list. These point into the data area. Whenever you change, say, the title of a card, the new title is written to the end of the data area and the index entry title pointer for that card is updated to point to the new location in the data area. Thus, in general, a notefile's data area grows every time any part of any card is changed. To throw away the old versions of card parts, it is necessary to compact the notefile.

1.2 Card IDs.

Every card in the notefile has a unique ID, e.g. NC00023. The top level fileboxes; **Contents**, **Orphans**, and **To Be Filed** have IDs NC00001, NC00002, and NC00003, respectively. Note that these boxes cannot be deleted. The IDs from NC00004 through NC00020 are unused. Currently, an old ID is never reused, even if its card is deleted and the notefile is compacted. Thus, if the inspector shows no entry in the card inspector menu for some ID, it is because that card has been deleted. If you've asked to show deleted cards and it still doesn't appear, it's because the notefile has been compacted since that card was deleted.

1.3 Card parts.

Of the four parts of a card, the title and property list are simplest. The title is simply a string while the property list is a list of attribute value pairs. If you have not been attaching properties to your cards, then the inspector will only show those properties that the system maintains. Currently the only such property is "Updates" with value equal to a list of dates on which the card was updated going chronologically backwards from the front of the list to the back.

The substance of the card is simply its contents. Thus a text card's substance is a text stream, a browser card's is its graph, etc. These are stored on the file in a manner appropriate to the substance type. Thus a text substance on the notefile looks like the way TEdit writes out text streams (text followed by "looks" information).

1.4 Links on the notefile.

The links of a card are divided into three groups: **to links**, **from links**, and **global links**, where the global links form a subset of the to links. The to links of a card are those links pointing from this card to some other card. The from links are those links pointing from another card to this one. Finally, the global links are those to links that are global, that is, they point from this card as a whole to another card. (Global links are the ones that aren't anchored in the source card's substance. Source links are an example.)

The confusing thing about links out on the notefile is that they are stored in several places. All links are stored as to links with their source card and as from links with their destination card. Furthermore, links that are not global are also stored within the substance of their source card inside of a link icon. Links that are global are also stored on the global links list of their source card. Thus, all links are stored in three places: as a to link on the source card, as a from link on the destination card and either in the substance of the source card or on its global links list. If these three records of a link don't agree for some reason, then we say that the notefile is **inconsistent** and needs its links rebuilt.

1.5 The links rebuilder.

The third phase of the inspector rebuilds the links of a notefile as follows: First it removes all the to and from links for every card. Then it reads the substances for each card and recreates to links and from links by looking at the links found inside the link icons in the substance.

The link rebuilder is also able to rebuild bad filebox substances. It does this by looking for all cards in the notefile with from links from the bad box and creating a new substance for the box containing only links to those cards. This process loses any text that the box might have contained as well as scrapping the original ordering of links. Nonetheless, in some cases this may be preferred to backing up the substance to a previous version or to deleting the box altogether.

The links rebuilder can rebuild the notefile's list of link types in a similar manner. That is, it records the set of link types seen on valid links and replaces the old links types with the new set. Note that this throws away any link types for which there are no links in the notefile.

Finally, the links rebuilder can rebuild bad global links for a card. It does this by looking for any cards with from links from the bad card that are global. This assumes that the card at the destination end has good links. Thus, if the cards at both ends of a global link have unreadable links, then there is no way to recover that link.

The inspector provides the option of having the links rebuilder phase rebuild bad filebox substances, bad link types, and bad global links. See Section ?? below.

2. Running the Notefile inspector: Phase 1: Scouring the data area

To start the inspector, first be sure that there is no open notefile. Then select the item **Inspect&Repair** from the NoteFile Ops menu. There is one option available at this level by "pulling to the side" called **ReadSubstances**. This ensures that substances of all cards pronounced valid by the inspector are readable. If this option is not invoked, then a check is still run on the length of the substance, but not on its contents. Unfortunately, the ReadSubstances option requires MUCH more work by phase 1. I recommend that you only use this option if Phase 3 (link rebuilding) breaks with some error like "Bad Piece Tbl" from TEdit. In that case, up-arrow out of the break and start the Inspect&Repair process over again, this time using the slower but more comprehensive ReadSubstances option.

Selecting Inspect&Repair will invoke phase 1 of the inspector, wherein the data area of the notefile is scoured for valid card parts. A record of all such parts is kept and statistics printed out at the end. You'll be asked to position the window in which those statistics as well as later inspector communications will be printed. You can monitor the progress of phase 1 by watching the prompt window. It will be printing messages like "Processing byte xxxxx of yyyyy."

When phase 1 has completed and you've positioned the interaction window, statistics on your notefile will be provided. You'll be told the total number of card IDs used and the number of those that are currently associated with active and deleted cards. (The rest are free and will never be reused. See Section 1.2 above.) If all seems well with the world, the next line will read "All active cards look okay." If not, there will be various messages outlining the problems. (See Figure 1.)

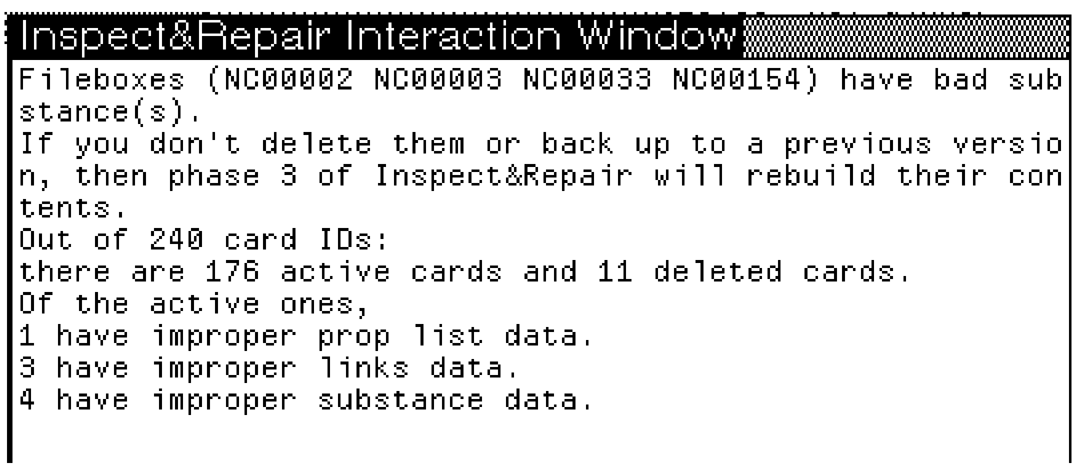


Figure 1: Snapshot of a sample interaction window

Note that several fileboxes have bad substances and that a special message is printed on their behalf. This indicates that if you don't wish to delete or back these up to a previous version, then phase 3 will rebuild them. (See Section 1.5 above.)

If there are cards having user-defined types whose type definition code has not been loaded, then you'll get a message to that effect, something like "<n> cards have unknown card types (FOO BAR)." At this point you should load the lisp files containing the definitions of the unknown card types. If not, then these cards will show up with bad substance in phase 2. If you have a card for which no substance versions could be read, then you'll also get unknown card type messages for it (reading something like "<n> cards have unknown card types (NIL)"). This is because the inspector couldn't find a card type on the notefile for that card.

A menu of options appears attached to the upper right corner of the interaction window. The particular options you get in that menu depend on the state of your notefile and are described below. The first two options appear in all cases. The other two may or may not be present in the menu you get. In any case, you should select one of the options before attempting any other NoteCards-related work.

ABORT: Choosing this option aborts the Inspect&Repair process entirely, throwing away any changes you might have made (such as card deletions or back ups.)

INSPECT CARDS: This brings up a menu of active card IDs with which you can inspect, delete, or back up particular cards. There is a "pull-across" menu item called INCLUDE DELETED CARDS, which if selected will include card IDs for deleted cards as well as active ones. Using this option, one can undelete deleted cards and restore some previous version.

END INSPECT&REPAIR: This option is only available if it seems that you don't need to continue to the link rebuilding phase. You will not get this option if you've deleted any cards, or generally if there are problems with the notefile. Choosing this option causes the Inspect&Repair process to end gracefully (via a normal checkpoint and close notefile), thus skipping phase 3, rebuilding links.

CONTINUE INSPECT&REPAIR: This option is only available if the notefile is in fairly good health (i.e. okay except for fileboxes to rebuild or global links to rebuild - see Section 1.5 above). Selecting it causes Inspect&Repair to move to phase 3 and rebuild your notefile's links.

3. Running the Notefile inspector: Phase 2: Your chance to tinker

After selecting INSPECT CARDS in the interaction window's attached menu, a menu containing Notecards IDs will pop up and be attached to the interaction window's lower left corner. It will contain IDs for all active cards and possibly deleted cards as well if you selected the submenu item INCLUDE DELETED CARDS described above. The menu can hold some 200 card IDs. If your notefile has more than that, then the menu will be composed of several pages each containing around 200 IDs. Rapid switching between pages is possible.

Attached to the upper right corner of the cards inspector menu is a menu containing at least the two options: ABORT and DONE. If the menu has multiple pages (there are more than 200 active cards in the notefile), then the attached menu will also include the items NEXT PAGE, PREVIOUS PAGE, and FIRST PAGE. Selecting these causes the current menu to be swapped with either the next menu, previous menu, or first menu, respectively.

Clicking ABORT causes the entire Inspect&Repair process to quit, throwing away any changes you've made. (This is equivalent to choosing ABORT from the inspector window as described in Section 2.)

Choosing DONE from this attached menu indicates that you're done tinkering with card parts and wish to return to the main interaction window. Normally, this causes the card inspector menu to close and the phase 1 process outlined in Section 2 to be performed again. Thus the data area will be rescoured and new statistics on the health of your notefile will be printed. This cycle of scour data area (phase 1) followed by inspect (phase 2) can be repeated as often as desired. Eventually, you must either abort, end the Inspect&Repair process gracefully, or continue to phase 3. Because phase 1 can be quite slow for large uncompacted notefiles, there is one optimization: if you've made no changes in phase 2, then the data area scouring is not repeated in phase 1. Rather, the old information from the last scouring is recovered and used instead.

If you've clicked DONE, but there are still cards with bad prop lists, titles, or links (because you haven't either deleted them or backed up their card parts to previous versions), you will be asked up to three questions. The list of card IDs of cards with bad prop lists is printed out and you're asked whether it's okay to set the prop lists of those cards to NIL. Then, the list of card IDs of cards with bad titles is printed out and you're asked whether it's okay to set the titles of those cards to "Untitled." Finally, card IDs for cards with bad links are printed out and you're asked whether it's okay to set the global links to nil. (Global links will be rebuilt as much as possible in phase 3. See Sections 1.5 and 4.) If you want phase 3 to be able to run, it is necessary to either answer yes to these questions or fix each of the bad card parts by hand in phase 2.

3.1 The card inspector menu

In the card inspector menu, those IDs corresponding to deleted cards have a line drawn through them. Those having some sort of problem appear shaded. In addition, an upper-case letter suffix is attached to such IDs indicating the problem. For example, a shaded menu item NC00023SL indicates that ID NC00023 has bad substance and bad links. The letter codes are S, L, P, and T indicating bad substance, links, property list, and title, respectively. If such a letter code appears in lower case, then the indication is that the current version of that card part is beyond the last checkpoint pointer. For example, NC00023t indicates that NC00023's current title was changed since the last checkpoint. (There may have been a crash, for example, thus preventing the notefile from closing normally.)

In addition to menu entries for each card ID, there is also one entry labeled LNTYPES allowing you to inspect (and possibly back up to a previous version) the Link Types for this notefile.

If you button an ID in the card inspector menu, then a popup menu allows up to two choices **Inspect** and/or **Delete**. If the card is currently deleted (has a line drawn through it), then the Delete option is replaced by **Undelete**. Certain card IDs cannot be deleted and thus their popup menus only contain the Inspect option. These are the top level file boxes NC00001, NC00002, NC00003. The link types menu entry LNTYPES also does not allow deletion.

Choosing Delete or Undelete from this popup menu causes the card to be deleted or undeleted, respectively and the line through the menu item either drawn or undrawn. Note, however, that this action (and all others) can be undone by choosing ABORT from either the card inspector menu or the interaction window menu.

Choosing Inspect from the popup menu for a card ID entry brings up a card parts inspector for that card.

3.2 The card parts inspector

Figure 2 below shows an example of a card parts inspector. It is composed of four attached menus arranged vertically and one attached operations menu atop the stack.

Figure 2: A card parts inspector

The four menus contain entries for every valid version of card parts for the card with ID NC00027. The top menu is for version of titles and below that are menus for versions of the card's substance, links, and prop list. For example, the Substance submenu contains entries for three versions of the substance of this card. The current version of each card part is shaded. Each menu entry gives the date that that version was written if available or the string "NO DATE AVAILABLE" if there is no date on the notefile. (The latter is the case for old notefiles prior to the time we began recording card parts write dates.)

If the current version of the card part is bad, then the menu entry will be a string so indicating, for example, "BADSUBSTANCE."

The title of the top menu includes the card's type and ID. In addition, each menu item contains a bit of information, in square brackets, before the date. In the title versions menu, this information is the first few characters of the title. In the substance versions menu, it is the number of bytes in the substance. In the links versions menu, it is a triple of numbers giving the number of to links, from links, and global links for this card. Finally, the proplist versions menu includes the number of entries on the property list for this card (i.e. twice the number of attribute-value pairs).

Atop the stack of menus is an attached menu of operations, described below.

ABORT: This aborts this card parts inspector, throwing away any changes made.

UPDATE: This closes the card parts inspector, effecting any changes (backing up to previous versions of card parts) you might have done.

DELETE: This option closes the card parts inspector and deletes the card. (Again, this can be undone by choosing ABORT from the card inspector menu.)

UNDELETE: For cards that have been deleted, this option appears instead of DELETE. Choosing it causes the card to be undeleted.

RESET: This causes the selections in the submenus to be restored to the values they had when the card parts inspector was first brought up. (Equivalent to doing ABORT and then inspecting this ID again from the card inspector menu.)

Note that cards that can't be deleted don't have the DELETE option on their card parts inspector.

Buttoning an entry in a submenu of a card parts inspector pops up a short menu unless the entry is for a bad version (e.g. "BADSUBSTANCE"). This menu contains at least the entry **Inspect** and possibly **Change Selection**, if the selected entry is not the same as the current one (i.e. not shaded).

Choosing Inspect allows further inspection of the details of the selected card part version. For example, inspecting a title version brings up the Interlisp inspector on a record containing the title, date and card ID. Similarly, inspecting a links or prop list version brings up the Interlisp inspector on a record containing the lists of links (for to links, from links and global links) or the prop list. Note that if you wish to continue inspecting a links version down to the single link level, choose to inspect the link as a NOTECARDLINK. This is somewhat more communicative about record field names. Note also that changing values out in the Interlisp inspectors has no affect on the notefile and is ignored.

All substance versions for cards having substance types TEXT, SKETCH and GRAPH can currently be inspected. (This includes all cards except those having user-defined substance types, like the NCFile card.) Inspecting a card's substance version will bring up a window showing a copy of the substance. (Note that changes to this copy have no affect on the notefile.) Any links in the substance of the card will show up as bracketed strings describing the link.

Choosing Change Selection from the card part version popup menu causes the current selection to be changed, thus backing up the card to the selected version. (This change can be undone by resetting or aborting the card parts inspector as well as by later aborting the card inspector or interaction window.)

4. Running the Notefile inspector: Phase 3: Rebuilding your links

To complete the Inspect&Repair process, select the **Continue Inspect&Repair** option from the interaction window menu. This invokes phase 3, the links rebuilder. Normally, this simply rebuilds links from card substances (see Section 1.5). In certain circumstances, it may do extra work as well. If your link types list is bad, and you didn't back it up to a previous version, then phase 3 will rebuild it. If there are fileboxes with bad substances that you haven't either deleted or backed up to previous versions, then phase 3 will rebuild them. Finally, if there are cards with bad links that you haven't backed up or deleted, then phase 3 will rebuild those links as well. (It rebuilds ALL to links and from links anyway. For those cards, it will rebuild global links as well.) Again, for details, see Section 1.5.

5. Tips and hints for using Inspect&Repair

This section contains a list of strategies and tips for using Inspect&Repair. For the most part, they are ordered from the useful and obvious to the esoteric. Several of these are implicit in the first four sections of the document, but are repeated here for emphasis and completeness.

When in doubt, abort! All your changes will be lost, but then if you're uncertain about what's happened this is the safe course. Often, in fact, you may simply want to check the health of your notefile and abort without tinkering.

Fixing versus tinkering. There are two main ways to use the inspector, either for fixing a broken notefile, or tinkering with a healthy one. The latter case occurs when you wish to recover some card that you inadvertantly deleted. Or back up a card that you inadvertantly changed to a previous version.

Compacting. Old versions of card parts have always been stored in notefiles, but up till now have been inaccessible. Thus, there was little reason not to compact your notefile often. Now there is a tradeoff between the need to save space by compacting versus the need to be able to back up using Inspect&Repair. Probably the safest course is to keep a backed up copy of the pre-compacted notefile around until you have confidence that the compacted one is healthy and that you have no need for previous versions of any of its cards.

Inspect&Repair can't run when notefile is open. This means that if you are working in your notefile and notice a card you'd like to inspect a previous version of, you must record the card's ID and close the notefile. Then, run Inspect&Repair, find the card ID in the inspector menu and tinker with it as desired.

Fixing enough problems to allow phase 3 to run. You can't run phase 3 unless Inspect&Repair thinks your notefile is above a certain threshold of health. There are certain problems it can handle (e.g. bad filebox substances, see Sections 1.5 and 4), and others that it can't (e.g. bad title). You have to decide either to fix these sorts of problems yourself in phase 2, let phase 3 attempt to rebuild them, or just abort the whole thing (always an option).

Sometimes these decisions can be tricky. For example, suppose a filebox's substance is bad. Call it BadBox. Should you (a) delete BadBox altogether, (b) back its substance up to a previous version, or (c) allow phase 3 to rebuild it by looking for from links in other cards from BadBox? Option (c) may not be advisable if there was important text in BadBox or if the order of cards in BadBox was important. On the other hand, option (b) may be of little use if the last good version is too out of date (or if there is no good version at all).

NoteCards Programmer’s Interface

Release 1.2i

Randy Trigg

Xerox PARC

Updated version: 18-Mar-85

Modified: 26-Aug-85 by Randy Trigg

Modified: 1-Aug-85 by Lissa Monty

Introduction

This document describes a facility whereby users with some programming know-how can obtain a lisp interface to NoteCards. In this way, they can create and modify Notefiles, cards and links under program control.

The functions described below are divided into 7 groups:

1. NoteFile Creation and Access

2. Creating and Accessing NoteCard Types

3. Creating NoteCards and FileBoxes

4. Accessing NoteCards and FileBoxes

5. Creating and Accessing Links

6. Creating and Accessing Link Labels

7. Handy Miscellaneous Functions

1.   NoteFile Creation and Access

For each of the following functions (except NCP.CloseNoteFile), the argument is a filename. The suffix ".NoteFile" is added if not already present. In any case, the filename used by NoteCards always has this suffix.

(NCP.CreateNoteFile <filename>)

If <filename> is not already a notefile, then create a notefile <filename>.NoteFile, and return this filename which can later be passed to NCP.OpenNoteFile.

(NCP.OpenNoteFile <filename> <don’tCreateFlg> <convertw/oConfirmFlg>)

If there is no currently open notefile, then open <filename> and make it the currently active NoteFile. Returns resultant stream if successful, else nil. If <don’tCreateFlg> is non-nil, then a new file will not be created if the given one doesn’t exist. If <convertw/oConfirmFlg> is non-nil, then if needed, the file will be converted to release1.1 format without user confirmation.

(NCP.CloseNoteFile [<stream>])

Closes <stream> if it is corresponds to a currently open Notefile. Returns its filename if successful. If <stream> is nil, then closes current open notefile.

(NCP.CheckpointSession)

Checkpoint the current Notecards session, first writing out any dirty cards. In case of a system crash or abort, the notefile can be recovered to the last checkpoint. Note that closing a notefile does a checkpoint.

(NCP.AbortSession)

Abort the current Notecards session, losing all work since last checkpoint or successful close.

(NCP.RepairNoteFile <filename>)

Rebuilds the link structure of <filename>. It must *not* be currently open.

(NCP.CompactNoteFile <filename>)

Copies <filename> to a later version, recovering space.   Must not be open.

(NCP.CompactNoteFileInPlace <filename>)

Compacts <filename> in place, replacing the old version.   Must not be open.

(NCP.DeleteNoteFile <filename>)

Removes the <filename> notefile. Must not be open.

(NCP.CurrentNoteFileStream)

Returns the currently open notefile stream if there is one, else nil.

(NCP.CurrentNoteFile)

Returns the full name of the currently active notefile if there is one, else nil.

(NCP.CheckOutNoteFile <fromFilename> <toFilename>)

Copies <fromFilename> to <toFilename> unless <fromFilename> is locked. If successful, creates a lock file in <fromFilename>'s directory. The name of the lock file is formed by concatenating the atom LOCKFILE onto <fromFilename>.

(NCP.CheckInNoteFile <fromFilename> <toFilename>)

Check lock file for <toFilename>. If none, then just copy <fromFilename> to <toFilename>. If there is one and it's owned by us, then do the copy and remove the lock file. If there is a lock file owned by someone else or if date of <toFilename> is more recent than date of lock file, then print a message and do nothing.

2. Creating and Accessing NoteCard Types

These functions give the user access to the NoteCard user-defined types facility. For an explanation of this facility, see the NoteCards Types Mechanism documentation.

(NCP.CardTypes)

(NCP.SubstanceTypes)

Returns lists of all currently defined NoteCard types and substances, respectively.

(NCP.CreateCardType <TypeName> <SuperType> <SubstanceType> <FnsAssocList> <VarsAssocList>)

Makes a new NoteCard type with name <TypeName>, super type <SuperType>, substance <SubstanceType>. Any functions not appearing in <FnsAssocList> will be inherited from <SuperType>. The CardWidth and CardHeight vars fields will be inherited if not specified in <VarsAssocList>. Other vars fields default to nil. Note that, for now, specializing the FileBox card type is not allowed.

(NCP.CreateSubstanceType <SubstanceName> <FnsAssocList> <VarsAssocList>)

Makes a new substance type with name <SubstanceName> and the given functions and vars fields. None of the function fields should be nil (but might conceivably be the function NILL).

(NCP.CardTypeSuper <type>)

Returns the super type of <type>.

(NCP.CardTypeSubstance <type>)

Returns <type>'s substance type.

(NCP.CardTypeLinkDisplayMode <type>)

Returns the link display mode of <type>.

(NCP.CardTypeFn <type> <fn>)

(NCP.CardTypeVar <type> <var>)

Returns the <fn> (<var>) field for <type>.

(NCP.CardTypeInheritedField <type> <field>)

Returns the value of the card type function or variable <field> for <type>. This is possibly different from the value returned by NCP.CardTypeFn or NCP.CardTypeVar in that if the defined value for <field> of <type> is nil, then the super is checked for a non-nil value. This checking continues until either a non-nil <field> is found or we reach the top of the super hierarchy. In that case, the value of <type>'s substance's <field> is used. Note that among the variable fields, only CardDefaultWidth and CardDefaultHeight inherit, so for the other Var fields, the result of NCP.CardTypeVar is valid (even if it's nil).

(NCP.SubstanceTypeFn <substance> <fn>)

(NCP.SubstanceTypeVar <substance> <var>)

Returns the <fn> (<var>) field for the substance <substance>.

(NCP.ValidCardType <type>)

Returns non-nil if <type> is an existing NoteCard type.

(NCP.ValidSubstanceType <substance>)

Returns non-nil if <type> is an existing NoteCard substance type.

(NCP.ValidCardTypeFn <fn>)

(NCP.ValidCardTypeVar <var>)

Returns non-nil if <fn> (<var>) is a valid function (variable) field for NoteCard types, for example, the litatom MakeCardFn (CardDefaultWidth). In other words, <fn> (<var>) can serve as the <fn> (<var>) arg to NCP.CardTypeFn (NCP.CardTypeVar).

(NCP.ValidSubstanceTypeFn <fn>)

(NCP.ValidSubstanceTypeVar <var>)

These return non-nil if <fn> (<var>) is a valid function (variable) field for substance types. In other words, <fn> (<var>) can serve as the <fn> (<var>) arg to NCP.SubstanceTypeFn (NCP.SubstanceTypeVar).

(NCP.CardTypeFns)

(NCP.CardTypeVars)

(NCP.SubstanceTypeFns)

(NCP.SubstanceTypeVars)

These return lists of all valid Fn (Var) fields for NoteCard types and substances respectively.

3. Creating NoteCards and FileBoxes

The following functions create various sorts of cards and boxes within the currently open notefile.

(NCP.CreateTextCard <title> <nodisplayflg> <props> <parentfileboxes>)

Creates and returns a new notecard having type Text. If <title> is non-nil, it is installed as the Notecard's title, otherwise the title is "Untitled." <props>, if non-nil, should be a prop-list of properties and values to be placed on the user property list of the Notecard. If <parentfileboxes> is non-nil, then it should be a list of FileBoxes in which to initially file this card.

(NCP.CreateFileBox <title> <nodisplayflg> <props> <childcardsboxes> <parentfileboxes>)

Creates and returns a new Filebox with title <title> (or a gensym'ed name if <title> is nil). It will initially contain child cards and boxes from the list <childcardsboxes> (if that arg is non-nil). If <parentfileboxes> is nil, then the new filebox will be filed in the value of (NCP.GetToBeFiledFileBox). The <props> arg is handled as it was for NCP.CreateNoteCard.

(NCP.CreateBrowserCard <title> <paramList> <nodisplayflg> <props> <parentfileboxes>)

Creates and returns a new browser card with given title, props and parents. <paramList> should be a prop list of browser parameters. The properties currently recognized are:

ROOTCARDS: A list of Notecards to serve as roots of the forest or lattice generated by the browser. If omitted or NIL then user is asked to choose root cards.

LINKTYPES: A list of link types to follow when creating the browser. Any label present in the list having the backarrow prefix ("_") represents that link type but in the reverse direction. This list can also contain the atoms ALL or _ALL in which case browsing will be done on all links in either the forward or reverse direction. If both ALL and _ALL are specified, then links in both directions will be used (generally making a mess).

DEPTH: The depth at which to cut off the browser. This should be a non-negative integer. If NIL or omitted, then will assume no limit. (Currently integers greater than 9 are assumed equivalent to infinity.)

FORMAT: This should be a list of one, two or three elements. The first should be an atom indicating grapher format. The choices are FAST (layed out as a forest, sacrificing screen space for speed), COMPACT (layed out as a forest, using minimal screen space), LATTICE (layed out as a directed acyclic graph, the default), *GRAPH* (layed out as a graph, i.e. virtual nodes are eliminated). The second element of the FORMAT list, if present, should be either HORIZONTAL (the default) or VERTICAL specifying whether the graph is layed on its side or up and down. The third element, if present, should be the atom REVERSE. This indicates that horizontal graphs should be layed out from right to left instead of left to right and that vertical graphs should be layed out from bottom to top rather than vice versa.

(NCP.CreateSketchCard <title> <nodisplayflg> <props> <parentfileboxes>)

Creates and returns an initially empty sketch/map card having given title, props, and parents.

(NCP.CreateGraphCard <title> <nodisplayflg> <props> <parentfileboxes>)

Creates and returns an initially empty graph card having given title, props, and parents.

(NCP.CreateCard <type> <title> <nodisplayflg> <props> <parentfileboxes> <otherargs>)

Creates and returns a card of the given (possibly user-defined) type, with given title, props, and parents. <otherargs> is a possibly nil list of args that will be passed to the MakeCardFn of <type>. Card is initially displayed or not according to value of <nodisplayflg>.

(NCP.MakeDocument <rootcard> <parametersProplist> <nodisplayflg> <props> <parentfileboxes>)

Creates and returns a Document card starting from <rootcard>. The default set of parameters for making documents can be accessed via NCP.DocumentParameters, but some of these can be given new values just for the duration of this MakeDocument by specifying a non-nil <parametersProplist>. For example, a value of '(TitlesFromNoteCards Bold ExpandEmbeddedLinks ALL) for <parametersProplist> would cause temporary changes to the values of the parameters TitlesFromNoteCards and ExpandEmbeddedLinks. As usual, the resulting card will have the given props and parents.

(NCP.MakeLinkIndex <linktypes> <backpointersP> <nodisplayflg> <props> <parentfileboxes>)

Creates and returns a LinkIndex text card consisting of a sorted record of all instances of links in the current notefile having one of the given link types. <linktypes> can contain the litatoms ALL and/or _ALL as well as any particular backwards links. (See the above description of NCP.MakeDocument.) Backpointer links are inserted in the text if <backpointersP> is non-nil. Resulting card will have given props and parents.

4. Accessing NoteCards and FileBoxes

The following functions provide access to the cards and boxes present in the current notefile. Note that whether a card's window has been brought up on the screen has little or no effect on the following functions. If the user changes some field of a card while that card is visible on the screen, then the field will update itself automatically. Thus, users can switch between program-driven and screen-interface-driven modes at will.

Cards can be active or inactive. An active card has its information cached (on its property list) thus saving time at the expense of memory. All cards visible on the screen are active. Most of the following functions leave the card in the same state as it was when they started (except NCP.BringUpCard, which makes it active). Thus, users needing to do several consecutive operations to the same card should consider temporarily caching the card's information via NCP.ActivateCards (and then uncache with NCP.DeactivateCards).

Most of the following functions take as first argument a card or filebox. If this does not in fact correspond to an existing card or box, then an error message is printed and nil is returned.

(NCP.BringUpCard <card> <region/position>)

Brings up on the screen the given card in the given region or at the given position. If <region/position> is nil, then user is asked to specify position with mouse.

(NCP.ActiveCardP <card>)

Returns non-nil if given card or box is currently active (i.e. information is currently cached in memory).

(NCP.ActivateCards <cardList>)

For each card or box in <cardList> (or just the one, if the argument is atomic), make it active (i.e. cache its information in memory).

(NCP.DeactivateCards <cardList>)

For each card or box in <cardList> (or just the one, if the argument is atomic), make it inactive (i.e. uncache its information back into the file). If any cards in <cardList> were on the screen then this will close their windows.

(NCP.CardType <card>)

Returns the type of <card> or NIL if the card does not exist.

(NCP.ValidCard <card>)

Returns non-nil if <card> exists (hasn't been deleted). (This is currently a synonym for NCP.CardType.)

(NCP.CardTitle <card> [<newtitle>])

Returns old title of <card>. If <newtitle> is present, then set <card>'s title to <newtitle>. <newtitle> can be an atom or string. Note, however, that all titles are converted internally to strings by NoteCards.

(NCP.FileCards <cards> <fileboxes>)

Every card or box in <cards> is filed in every box in <fileboxes>. Either arg may be an atom or a list.

(NCP.UnfileCards <cards> <fileboxes>)

Every card or box in <cards> is unfiled from every box in <fileboxes>. Furthermore if <cards> is the litatom ALL, then the boxes in <fileboxes> will be cleared of all children. Similarly, if <fileboxes> is the litatom ALL, then the cards and boxes in <cards> will be unfiled from all their parent boxes. Either arg may be an atom or a list.

(NCP.CardParents <card>)

Returns list of fileboxes in which <card> has been filed.

(NCP.FileBoxChildren <filebox>)

Returns list of children of <filebox> in the order in which they appear in the box's textstream.

(NCP.GetLinks <cards> <destinationCards> <labels>)

Returns list of all links from any of <cards> to any of <destinationCards> having any label in <labels>. Any of these arguments can be nil. For example, if <destinationCards> is nil, then all links pointing from <cards> to anywhere with a label in <labels> are returned. If both <cards> and <destinationCards> are nil, then this returns all links having a label in <labels>. If all three args are nil, then this is a slow synomym for NCP.AllLinks.

(NCP.CardPropList <card>)

Returns the prop list of the given card.

(NCP.CardProp <card> <propname> [<newvalue>])

Returns old value of property <propname> on <card>'s prop list. If <newvalue> is present, then set <card>'s <propname> property to <newvalue>. (Semantics are analogous to the Interlisp function WINDOWPROP.)

(NCP.CardAddProp <card> <propname> <newitem>)

Adds <newitem> to the list present on the <propname> property of <card>. Returns old value of property. (Semantics are analogous to WINDOWADDPROP.)

(NCP.CardDelProp <card> <propname> <itemToDelete>)

Deletes <itemToDelete> from the <propname> property of <card> if it is there, returning the previous value of that property. If not there, return nil. (Semantics are analogous to WINDOWDELPROP.)

(NCP.CardSubstance <card>)

Returns the substance of <card>. This is a textstream in the case that the type of <card> has TEXT substance. Otherwise, it is the appropriate underlying structure if <card> has GRAPH or SKETCH substance.

(NCP.CardRegion <card>)

Returns the region of <card>. This works even if <card> is not currently up on the screen, since the region information is stored on the notefile.

(NCP.CardAddText <card> <textstr> <loc>)

Adds the text within the string <textstr> to the text card <card>. If <loc> is the litatom START or END, then the text will be placed at the start or end of the card respectively. If <loc> is a number, then it is assumed to be a character count within the card at which to place the new text. If <loc> is NIL, then the text is placed at the current cursor location.

(NCP.ChangeLoc <card> <loc>)

Changes the cursor's location in <card>'s textstream to <loc>. Possible values for <loc> are as described for NCP.CardAddText.

(NCP.DeleteCards <cards>)

Deletes the given cards and fileboxes from the current notefile, or deletes just the one if <cards> is atomic.

(NCP.FileBoxP <card>)

Returns non-nil if <card> is a filebox.

(NCP.AllCards)

Returns a list of all extant cards for the current notefile.

(NCP.AllBoxes)

Returns a list of all fileboxes in the current notefile.

(NCP.MapCards <fn>)

Maps down the set of all cards in the current notefile, applying <fn> to each.

(NCP.MapBoxes <fn>)

Maps down the set of all fileboxes in the current notefile, applying <fn> to each.

(NCP.GetContentsFileBox)

(NCP.GetOrphansFileBox)

(NCP.GetToBeFiledFileBox)

These functions retrieve the three predefined FileBoxes for the currently open NoteFile. These boxes can be modified (but not deleted) by the user in the same way as any other filebox.

5. Creating and Accessing Links

Links can be connected to points within a card or to the card as a whole, thus the following four link creation functions are provided. Those that connect to points within a card specify at least one of <fromloc> or <toloc>. If nil, then the link icon is placed at the current cursor location in the card. If the arg is the litatom START or END, then it is placed at the front or end of the text respectively. If the loc arg is a number, then it is assumed to be a character count at which to place the link icon.

(NCP.GlobalGlobalLink <label> <sourceCard> <destinationCard>)

Creates and returns a new link with label <label>, connecting <sourceCard> to <destinationCard>.

(NCP.LocalGlobalLink <label> <sourceCard> <destinationCard> <fromloc> <displaymode>)

Creates and returns a new link with label <label>, connecting from <fromloc> of <sourceCard> card to <destinationCard>. If <displaymode> is non-nil, then the new link is displayed in the given mode. Otherwise the default displaymode for the source card's type is used.

(NCP.GlobalLocalLink <label> <sourceCard> <destinationCard> <toloc>)

Not implemented at this time.

(NCP.LocalLocalLink <label> <sourceCard> <destinationCard> <fromloc> <toloc>)

Not implemented at this time.

(NCP.LinkDesc <link>)

Returns list of three items (<label> <sourceDesc> <destinationDesc>) where <label> is the link type and <sourceDesc> and <destinationDesc> have the form (<anchor mode> <card> <loc>). <anchor mode> is either LOCAL or GLOBAL, <card> is the card at this end of the link, and <loc> gives a position in the text of <card> if <anchor type> is LOCAL and <card>'s substance's type is TEXT.

(NCP.LinkDisplayMode <link> [<newdisplaymode>])

Returns old diplay mode of <link>. If <newdisplaymode> is present, then set <link>'s displaymode accordingly. If non-nil, it can be one of the litatoms Icon, Title, Label, or Both. Or it can be an instance of the LINKDISPLAYMODE record. This has the 3 fields SHOWTITLEFLG, SHOWLINKTYPEFLG, and ATTACHBITMAPFLG. Each field can have one of the three values T, NIL, or FLOAT. If a field, say SHOWTITLEFLG, has value FLOAT then the corresponding global parameter (DefaultLinkIconShowTitle, in this case) will be consulted to decide whether or not to display the destination card's title in this icon. (See Section 7 for a description of the global parameters.)

(NCP.LinkLabel <link> [<newlabel>])

Returns old label of <link>. If <newlabel> is present, set <link>'s label to <newlabel>.

(NCP.GetLinkSource <link>)

Returns the card at the source end of <link>.

(NCP.GetLinkDestination <link>)

Returns the card at the destination end of <link>.

(NCP.DeleteLinks <links>)

Removes all links in <links> (or the single one if <links> is atomic).

(NCP.ValidLink <link>)

Returns non-nil if <link> is a link in the current notefile.

(NCP.AllLinks)

Returns a list of all existing links in the current notefile. (This is equivalent to but faster than (NCP.GetLinks NIL NIL NIL).)

(NCP.MapLinks <fn>)

Maps down the set of all links in the current notefile, applying <fn> to each one.

6. Creating and Accessing Link Labels

The following functions allow the user to manipulate link labels.

(NCP.CreateLinkLabel <label>)

Creates a new link label with name <label> for current notefile unless there is already one defined by that name.

(NCP.DeleteLinkLabel <label>)

Deletes the link label <label> from the current notefile. The label must exist and must not be the label of any existing link, and it must not be a system-defined link label (e.g. SubBox or FiledCard).

(NCP.RenameLinkLabel <label> <newlabel>)

Changes any links having label <label> to have label <newlabel>. <label> must exist and neither <label> nor <newlabel> should be a system-defined label.

(NCP.GetLinkLabels)

Returns a list of all existing link labels including system-defined ones.

(NCP.GetReverseLinkLabels)

Returns a list of the reverse labels for every existing link label. Thus, whereas SubBox would appear in the list returned by NCP.GetLinkLabels, _SubBox would appear in the list returned by NCP.GetReverseLinkLabels.

(NCP.GetUserLinkLabels)

Returns a list of all existing user-defined link labels.

(NCP.ValidLinkLabel <label>)

Returns non-nil if <label> is a defined link label for current notefile.

7. Handy Miscellaneous Functions

(NCP.TitleSearch <key> <key> ...)

Returns a list of all cards having all of the <key>s (can be atoms, numbers or strings) within their titles.

(NCP.PropSearch <propOrPair> <propOrPair> ...)

Returns a list of all cards such that for every <propOrPair> arg, if it is atomic, then the card must contain that property. If it is a list of two elements, then the card must have a property EQ to the first element with value EQ to the second element.

(NCP.WhichCard <x> <y>)

Returns the card currently displayed on the screen whose window contains the position in screen coordinates of <x> if <x> is a POSITION, the position (<x>,<y>) if <x> and <y> are numbers, or the position of the cursor if <x> is NIL. Returns NIL if the coordinates are not in the window of any card. If they are in the window of more than one card, then returns the uppermost. If <x> is a window, then NCP.WhichCard will return the card associated with that window.

(NCP.CardFromWindow <window>)

Returns the card associated with <window>, or NIL if not a notecards window.

(NCP.CardWindow <card>)

Returns <card>'s window if <card> is currently displayed somewhere on the screen.

(NCP.SelectCards)

Returns a list of those cards selected from the screen. A menu appears near the top of the screen with buttons for "DONE" and "CANCEL". Selections are made by left buttoning in the title bars of the desired cards.

(NCP.DocumentParameters <parametersProplist>)

Returns the old value of the document parameters in the form of a proplist. If <parametersProplist> is non-nil then it should be a proplist whose properties are (some of the) valid document parameter names and whose values are permissible values for those parameters. The valid parameters and possible values are as follows:

HeadingsFromFileboxes: NumberedBold, UnnumberedBold, NONE.

TitlesFromNoteCards: Bold, NotBold, NONE.

BuildBackpointers: ToCardsBoxes, ToCards, ToBoxes, NONE.

CopyEmbeddedLinks: ALL, NONE, <listOfLinkLabels>.

ExpandEmbeddedLinks: ALL, NONE, <listOfLinkLabels>.

[See the Notecards user's manual for an explanation of these parameters and how their values affect the document created.]

(NCP.NoteCardsParameters <parametersProplist>)

Returns the old value of the global Notecards parameters in the form of a proplist. If <parametersProplist> is non-nil then it should be a proplist whose properties are (some of the) valid document parameter names and whose values are permissible values for those parameters. The valid parameters and possible values are as follows:

DefaultCardType: <legalCardType>

FixedTopLevelMenu: T or NIL

ShortWindowMenus: T or NIL

ForceSources: T or NIL

ForceFiling: T or NIL

ForceTitles: T or NIL

CloseCardsOffScreen: T or NIL

MarkersInFileBoxes: T or NIL

AlphabetizedFileBoxChildren: T or NIL

DefaultLinkIconAttachBitmap: T or NIL

DefaultLinkIconShowTitle: T or NIL

DefaultLinkIconShowLinkType: T or NIL

LinkDashingInBrowsers: T or NIL

ArrowHeadsInBrowsers: one of the litatoms {AtEndpoint, AtMidpoint, None}

SpecialBrowserSpecs: T or NIL

AnnoAccessible: T or NIL

EnableBravoToTEditConversion: T or NIL

DefaultFont: a font

LinkIconFont: a font

Here, <legalCardType> should be an existing Notecard type, i.e. one that appears in the list returned by NCP.CardTypes.

(NCP.PrintMsg <window> <clearFirstFlg> <arg1> <arg2> ...)

Prints a message in the prompt window of <window>. If <window> is NIL, then prints message in the Lisp prompt window. If <clearFirstFlg> is non-nil, then clears the prompt window first. The args are PRIN1'ed one at a time.

(NCP.ClearMsg <window> <closePromptWinFlg>)

Clears the prompt window associated with <window> (or with the main Lisp prompt window if <window> is NIL) and closes it if <closePromptWinFlg> is non-nil.

(NCP.AskUser <Msg> <prompt> <FirstTry> <ClearFirstFlg> <MainWindow> <DontCloseAtEndFlg> <DontClearAtEndFlg> <PROMPTFORWORDFlg>)

This function can be used to ask questions of the user in a window's prompt window. The <Msg> and <prompt> are printed along with <FirstTry> (if non-nil). The value returned is whatever the user types. If <ClearFirstFlg> is non-nil, then the prompt window is cleared first. If <MainWindow> is nil, then the top level prompt window is used. If <DontCloseAtEndFlg> is non-nil, then the prompt window won't be closed after the question is answered and if <DontClearAtEndFlg> is non-nil, then the prompt window won't be cleared at the end. If <PROMPTFORWORDFlg> is non-nil, then the PROMPTFORWORD typein protocol will be used rather than TTYIN. The former doesn't allow mouse editing of the string typed in. On the other hand, typing automatically overwrites the prompt when PROMPTFORWORD is used.

(NCP.AddTitleBarMenuItems <Win> <NewMenuItems>)

Adds the given menu items to the left button title bar menu of <Win>. <Win> should be the window of a visible notecard.

(NCP.GetDates <Card>)

Returns a NOTECARDDATES record structure containing the dates of last modification of each of the four card parts of <Card>. The fields of the record are SUBSTANCEDATE, TITLEDATE, LINKSDATE and PROPLISTDATE.

Notice of release of Notecards 1.2i

The 1.2i Intermezzo release of Notecards is hereby officially released.

To run NoteCards, load onto an Intermezzo sysout the file {qv}<notecards>release1.2i>notecards.dcom. As usual, send bug reports by choosing "NoteCards Report" from the Lafite middle button send mail menu. Mail of more general interest to the NoteCards community should be sent to NoteCards^.pa.

Even if you have been using 1.2i for some time and feel comfortable with Notecards, please take a look at the release notes in {qv}<notecards>release1.2i>doc>ReleaseNotes.ted.

The release notes describe in detail the changes since 1.1. These include incorporation of the latest version of sketch, new functionality in the browser, a new notefile inspect and repair facility, several new library packages, and many other feature additions and bug fixes.

Also on {qv}<notecards>release1.2i>doc> you can find updated documentation on the programmer's interface, ProgIntFace.ted, and a new document describing the inspect and repair facility, NoteFileInspector.ted.

The library packages and accompanying documentation can be found in {qv}<notecards>release1.2i>library>.

Enjoy!

- Randy

NoteCards Release1.2i Announcement

Xerox Corporation

Randy Trigg
Frank Halasz

[Location: {qv}<notecards>release1.2i>notecards.dcom]
[First written: 3/27/85 Randy Trigg]
[Last updated: 8/26/85 Randy Trigg]

This document updates the NoteCards Release1.1 User's Manual, describing changes and new features for Release1.2i. As usual, send bug reports to NoteCardsSupport.pa (or use the Lafite SendMail middle button menu) and matters of more global interest to Notecards^.pa.

You must be in Intermezzo to run NoteCards Release1.2i. From now on, you can depend on the letter suffix following the release number to indicate the appropriate version of Interlisp.

Changes from 1.1 are mostly in the following areas: the NoteCards browser, notefiles interface, link icon display and user interface. In addition, there are various miscellaneous changes, a couple of new card types, and fixes of several outstanding 1.1 bugs.

1. Operating on a Notefile.

Checkpointing and aborting a session:

A fundamental change was made to the way Notecards updates its working notefile that allows 1.2i users to checkpoint their work, abort a session (losing work since the last checkpoint), and recover more gracefully from crashes. First, a word about the way Notecards notefiles are structured.

A notefile consists of two parts, an index area and a data area. The index includes for each notecard, several pointers into the data area. There are separate pointers for the notecard's substance, title, prop list, and links. When, say, a notecard's title is changed, the new title is written at the end of the data area (in fact the end of the file) and the index pointer is changed. In Release1.1 (and earlier), the index modifications happened out on the file as they occured. Now, in Release1.2i they happen in an in-core array and are not written to the file till checkpoint (or close) time. In addition, there is a checkpoint pointer that points to the end of file at the time of the last checkpoint or close. New data (such as a new title) is still written to the file, but always at the end of the file. Thus if a crash occurs and later the notefile is reopened, Notecards can notice the extra data beyond the checkpoint pointer and truncate the file at that point (if you confirm).

More concretely, there are now two new NoteFile Ops menu entries: "Checkpoint Session" and "Abort Session." Checkpointing causes any active cards to have their contents saved to the notefile (but not closed), the index array to be written back out to the file, and the checkpoint pointer to be reset to the end of the file. (Note that closing a notefile automatically does a CheckpointSession.) Aborting a session causes Notecards to close down, discarding all work since the last checkpoint or close.

When a notefile is opened, the checkpoint pointer is compared with the end of file pointer. If they don't agree, then you're asked whether the file should be truncated. You're also given the option of saving the extra work since the last checkpoint to a file. If valuable cards were created (or modifications made) since the last checkpoint, then you should answer yes and provide the name of a file in which to store the truncated information.

Next, you should open the truncated notefile and bring up a separate TEdit window on the file containing the truncated information. Though TEdit formatting information is lost, you can recover a card's text by browsing this file. (Note that scrolling from back to front will retrieve the most recent version of each card.)

[Note that closing (or saving without closing) a card writes it out to the file, but does not force the index to be updated. Thus, if crashes are anticipated, do CheckpointSession often.]

Compacting a Notefile:

Because Notecards never actually overwrites any information in the data area of a notefile, it is necessary to periodically compact the notefile. This facility has been improved in Release1.2i in two ways. It is now possible to specify a target file name for the compaction (rather than always going to the same name), and it is now possible to compact a notefile in place. These two choices form a submenu of the CompactNotefile entry in the Notefile Ops menu.

Copying, restoring, and backing up notefiles:

The menu entries for RestoreFromFloppy and BackupToFloppy have been removed from the NotefileOps menu. In their place is a general CopyNotefile option. It prompts you for source and target file names for the copy.

There is a new facility for checking in and out notefiles using locks for multiple users sharing a notefile. Still in the experimental stages, it must be called via the programmer's interface. See the programmer's interface documentation.

Inspecting and healing broken notefiles:

The old Repair option on the Notefile Ops menu is now called Inspect&Repair and has been improved considerably. Before rebuilding the links of your notefile, it reads the entire data area looking for good card parts (including outdated and deleted versions). It then allows you to delete and/or back up card parts to previous versions. All this is done interactively through a menu driven interface. Only when the notefile is deemed healthy are you allowed to perform the link rebuilding. For details on the operation of Inspect&Repair, see the document titled NotefileInspector.ted.

2. Changes to the Notecards user interface.

Stylesheets:

Several places in Notecards now use Tayloe Stansbury's stylesheet package for user interaction, in particular, changing a link's display mode, a browser's specs, or the default text and link icon fonts from the global parameters menu. Stylesheets allow packaging of several menus together with "buttons" governing individual menus and the stylesheet as a whole. Menus within a stylesheet can optionally allow multiple selections. All stylesheets have three global buttons "Done," "Reset," and "Abort." "Done" causes the new values to be accepted. "Reset" causes the original values (when the stylesheet was entered) to be recovered. "Abort" causes the stylesheet to be exited without changing any values. Menus allowing multiple selections also have the buttons "All" and "Clear" attached. "All" causes all values in the menu to be selected while "Clear" unselects the entire menu. Toggling of menu entries is accomplished by left clicking the entry.

New global parameters:

The top level global parameters menu has several new additions. These (as well as some old 1.1 ones) are described below. To change the value of a global parameter, click on the variable name. The value will toggle between "Yes" and "No" if binary, and allow selection from an appropriate menu otherwise.

ForceSources, ForceFiling, ForceTitles: These dictate whether to bother you at card closing time about incomplete information for the card. If ForceFiling is set, for example, then you are asked to designate parent fileboxes of the card before closing. Similarly, for sources and titles. If ForceFiling is off (value is "No"), then cards without parents will be filed automatically in the ToBeFiled filebox at closing time. If ForceTitles is off, then an untitled card will be left with the title "Untitled." ForceTitles and ForceFiling default to "Yes," while ForceSources defaults to "No."

CloseCardsOffScreen: If "Yes," then when a card is closed, it is first dragged off screen so that the close happens invisibly.

MarkersInFileBoxes: If "Yes," then new fileboxes will contain the markers "FILE BOXES" and "NOTE CARDS." New child boxes are inserted under the FILE BOXES marker and new child cards under the NOTE CARDS marker. If "No," then new fileboxes come up without markers and new children are inserted at the current cursor position. Note that regardless of the MarkersInFileBoxes setting, if a filebox has no markers (because you've deleted them) then new children are inserted at the cursor position.

AlphabetizedFileBoxChildren: If "Yes," then new fileboxes will have the property OrderingFn set to NC.IDAlphOrder. This will cause any new cards put into such a filebox to be inserted in alphabetical order. For further details on OrderingFn's for fileboxes see Section 4.

DefaultLinkIconAttachBitmap, DefaultLinkIconShowTitle, DefaultLinkIconShowLinkType: These dictate the manner in which link icons are displayed if not currently specified in the icon. There are three fields of a link's display mode that can be set, unset, or floated independently. If a field is floated, then the global parameter for that field is consulted. For example, if a link icon's display mode has value FLOAT for the ShowTitle field, then whether the title gets shown inside the link icon depends on the value of DefaultLinkIconShowTitle. See below for a further description of a link's display mode.

LinkDashingInBrowsers: If "Yes," then browser links are drawn with dashed lines with the dashing style corresponding to the link's type. See Section 3 for further details on browser changes. Defaults to "No."

ArrowHeadsInBrowsers: This dictates whether arrow heads are drawn on browser links. The variable can be set to either "AtMidpoint," "AtEndpoint," or "None." See Section 3 for details. Defaults to "None."

EnableBravoToTEditConversion: If "Yes" then TEdit checks when getting a file whether that file is in Bravo format and if so, converts. This defaults to "No" for efficiency.

DefaultFont: This dictates the font that new text cards default to.

LinkIconFont: This dictates the font for text appearing in link icons.

Link icon display mode:

The display mode of a link icon can be changed by middle buttoning in the icon and selecting from the three menus in the resulting stylesheet. These are: AttachBitmap, ShowTitle, and ShowLinkType. AttachBitmap, if "Yes," causes link icons to be shown with a bitmap representing the type of the destination card attached at the left. ShowTitle and ShowLinkType, if "Yes," cause the link icon to contain the title of the destination card and/or the link type. Any of the three fields can have the value FLOAT, in which case the appropriate global parameter will be consulted. (See description of global parameters above.) If all fields are set to "No" (or the floating ones inherit No from the global parameters), then a small, uninformative icon is used to display the link.

"Pushing" and "Pulling" link icons:

There are now two ways to move or copy a link icon between cards or within a card. "Pulling" works like TEdit shift-select. That is, to move an icon, put the cursor where you want to move to and hold down the shift key (or shift and ctrl keys) while left clicking in the left or right quarter of the icon. The new style is called "pushing" and is done by holding down the shift key while left clicking in the middle part of the icon. Then move the cross-hairs cursor to the icon’s new home and left-click. To abort a "push," just left click in the background. Note that "pushing" currently only works for copying, not moving.

Specifying notefile names and card titles:

A different editor has been incorporated into Notecards for obtaining card titles, file names, etc. This editor is the same one used in the top level lisp exec window (TTYIN). Thus you can change the title (or file name) given as prompt via mouse edits.

3. Changes to the Notecards browser.

Multiple roots:

Browsers can now contain multiple roots, in which case the graph will be laid out as a forest.

Dashed links:

Dashed browser links was a rarely used option in Release1.1, largely because of speed considerations. The speed of drawing dashed links has improved in Release1.2i by taking advantage of improvements in Grapher. There are currently nine different dashing styles possible. If a browser contains instances of more than nine different link types, then the last dashing style will be used repeatedly for each link type beyond the ninth. As before, link dashing is a user-settable option in the GlobalParameters menu (see Section 2).

Arrowheads:

Arrowheads can now be drawn on browser links. These show the direction of the notecards link being represented in the browser. This is a user-settable parameter in the GlobalParameters menu with possible values AtMidpoint, AtEndpoint, or None. If AtMidpoint or AtEndpoint is specified, then arrowheads will be drawn at link midpoints or endpoints, respectively. However, in either case, if two browser nodes are connected by more than one link, then any arrowheads for those links will appear at the midpoints (so as not to overlap).

Browser specs:

Whereas in Release1.1 only the link types to traverse could be specified, in Release1.2i, link types is one of a number of browser specs. Also included are browser depth, format, and orientation. These are accessible through a BrowserSpecs stylesheet, a collection of 5 menus. For general details on the stylesheet interface see Section 2. In this case, the forward and backward link types menus are multi-selectable, that is, more than one entry can be chosen. The other three menus are used to make single selections.

Forward and backward link types function as in Release1.1. That is, the browser will contain only nodes for cards reachable from the root cards by following forward links in "line of direction" or backward links in "reverse line of direction."

Browser depth is chosen from a menu containing entries for the integers 0 through 9 and INF (or infinite depth). The default is INF, meaning that the browser will not be cut off until there are no more links to follow from leaf nodes. Choosing depth 0 means that only the root nodes will appear (and no links).

Browser format is one of *GRAPH*, LATTICE, COMPACT, or FAST. The latter three are provided by the grapher package and correspond to lattice, compact forest and fast forest, respectively. COMPACT and FAST generate virtual nodes (in double boxes) whenever two or more links would be drawn to the same node. LATTICE only generates virtual nodes when a cycle exists in the graph. *GRAPH* is a new format that never generates virtual nodes. The drawback to using *GRAPH* is that a cycle can cause lines to be drawn that cross boxes or overlap other lines. Thus you may have to move nodes around for legibility after computing the browser. The default is LATTICE.

Browser orientation is one of Horizontal, Vertical, Reverse/Horizontal, or Reverse/Vertical. These specify whether the graph is layed out left-to-right, top-to-bottom, right-to-left, or bottom-to-top, respectively. The default is Horizontal.

New middle button title bar menu options:

Several new entries have been added to the middle button menu invoked from a browser's title bar. The options are now RecomputeBrowser, RelayoutGraph, ReconnectNodes, UnconnectNodes, ExpandBrowserNode, GraphEditMenu, and ChangeBrowserSpecs.

RecomputeBrowser causes the current contents of the browser to be thrown away and recomputed as in Release1.1. However, in Release1.2i, you can optionally specify a new set of root nodes.

RelayoutGraph does not rebuild the graph, but rather causes the nodes and links of the graph to be repositioned on the screen (using Grapher's LAYOUTGRAPH). This will destroy any work you have done moving nodes within the graph.

ReconnectNodes first causes any link edges in the graph to be erased. (Note, however, that non-link edges, those created by "AddEdge" as described below, are ignored.) Then, each node in the graph is connected to every other node in the graph for which there is a link between them having one of the currently selected link types. This can be useful for several reasons:

1. when the linking structure between cards has changed, but the current browser layout needs to be preserved.

2. when some browser nodes need to be moved, but dragging the connected links is too slow. In this case, do UnconnectNodes followed by ReconnectNodes (after you've moved the nodes around).

3. when special browser layouts are desired. For example, suppose you like the layout that Grapher gives you when certain links are left out or when you limit the depth. Then calling ReconnectNodes will fill in the missing links without affecting the graph's layout.

UnconnectNodes simply erases all edges in the browser. This is useful for positioning a browser's nodes before invoking ReconnectNodes.

ExpandBrowserNode allows you to enlarge the graph under a given node. After selecting a node, you're asked for a depth (defaults to 1). The graph is then expanded under the selected node to the given depth, following any currently selected links. Note that ExpandBrowserNode calls LAYOUTGRAPH so any existing special node arrangements will be lost.

GraphEditMenu brings up the graph editing menu. See the description below.

ChangeBrowserSpecs brings up the BrowserSpecs stylesheet to allow you to change any of the browser specs. These changes will be noticed at the next RecomputeBrowser, ExpandBrowserNode, etc.

Editing the browser manually and "structure editing":

The browser can be edited through the use of the GraphEditMenu. This menu can be obtained either by right-buttoning in the browser window or by choosing GraphEditMenu from the title bar middle button menu. The GraphEditMenu includes options for "structure editing"; that is, changing underlying NoteCards structure by editing the browser. The old options for editing without changing structure are also present. Given below are the menu items in GraphEditMenu and the actions they engender.

CreateCard&Node causes a new card to be created in the current Notefile and a corresponding node for it to be included in the browser. You're asked for the type of the new card, its title, and where to position the node representing it.

CreateLink&Edge causes a new link to be created between two existing cards and a corresponding edge to be drawn in the browser. (We call such an edge representing a Notecards link, a "link edge." See AddEdge below for creating non-link edges.) You're asked for the "From" and "To" nodes in the browser corresponding to the cards to be linked as well as a link type. The link icon for the new link is positioned at the cursor point in the From card if the card has text substance and an open window. Text cards with closed windows have links inserted at the start of the text stream. Otherwise, the new link is a global link. You can have multiple link edges between pairs of cards. In this case the edges are displayed in a spline or "flower" arrangement.

DeleteCard&Node causes a card to be deleted and its corresponding node in the browser to be removed. You are asked first to choose the node representing the card to be deleted and then to confirm the removal of the node (type "y" to confirm) and the deletion of the card. If the selected node is one of a set of virtual nodes (double boxed), then all nodes in the set (i.e. representing the given card) are removed.

DeleteLink&Edge causes a link in the Notefile to be deleted and the corresponding edge in the browser to be removed. You first pick the "From" and "To" nodes corresponding to the source and destination ends of the link respectively. Then, if there is only one link between those two cards, the link is deleted after user confirms. If there are multiple links between the two cards, then the user chooses from a menu of link types.

AddLabel puts a "label node" into the browser that does not represent a Notecard. You are prompted for a string forming the node's label and then must position the label node. This node is not boxed. (But note that "virtual" label nodes can be boxed and thus can be confused with non-virtual regular nodes.)

AddNode adds a node into the browser corresponding to some existing card. You are asked to point to a card (title bar or link icon) on the screen that this node is to represent and then to position the node.

AddEdge draws a line between two nodes in the browser. This edge does not correspond to a real link in the Notefile. To avoid confusion, it is best to have the arrowheads option on (see Section ??) in this case, since edges formed by AddEdge do not have arrowheads (or dashing). Only one such edge is allowed between any two nodes and none if there are already link edges between the nodes. Thus doing CreateLink&Edge will remove any existing non-link edge.

RemoveNode removes a node from the browser. It does not delete the card (if any) that the node represents. Edges into and out of the node are also removed. If the selected node is one of a set of virtual nodes representing the same card, then you will be told how many nodes will be removed with this one and will be asked to confirm. The only way to remove only one node of a set of virtual nodes, is to first manually remove edges into and out of it using RemoveEdge. Then RemoveNode can be used to remove only the one virtual node.

RemoveEdge removes an edge from the browser. It does not delete the link (if any) that the edge represents. The user is asked to select the "From" and "To" nodes of the edge.

MoveNode allows you to change the position of any node, rubber banding any edges pointing to it. You're asked to point to the node by left-buttoning, and holding down the left button, drag the node to its new position.

LabelSmaller is used to decrease the font size of label nodes. Note that it does not work for regular non-label nodes.

LabelLarger is used to increase the font size of label nodes.

<->Shade toggles the shade of a node between black-on-white and white-on-black. This can only be performed on label nodes (not on nodes representing Notecards).

FIX MENU causes the GraphEditMenu to be affixed to the lower right edge of the browser window. Note that this does not prevent you from obtaining the menu via right button inside the window.

[Note that the above editing commands do not work on old 1.1 browsers. Such browsers should either be recomputed (via RecomputeBrowser) or unconnected and reconnected.]

4. Miscellaneous changes.

Links ordering within text cards:

The internal list of outgoing links in a text card is now kept in the same order that the links appear in the card's text. This means, for example, that the daughters of a browser node for a filebox will appear in the correct order.

Link insertion:

The title bar menu entry for "InsertLinks" now has an attached submenu containing entries for adding single links, multiple links, and global links. When inserting multiple links (or adding multiple global links) you're only asked for one link type which is used to label all the new links and all are inserted at the same place in the text.

Show links:

This is now a normal entry in the left button title bar menu of a card (rather than a subentry under Edit Properties). The format of the ShowLinks display has been changed slightly. The prefix is now either TO, FROM, or Global TO. The link type is shown in the icon. Also, for text cards, the TO links should appear in the correct order.

Sketch changes and fixes:

Notecards now uses the latest version of sketch. See the sketch documentation for details on changes. Several long-standing bugs having to do with link icons in sketch cards have been fixed.

Sketches and graphs in text cards:

It is possible to shift-copy the contents of sketch and graph/browser cards into text cards. In addition, the Document card is now able to include the contents of sketch and graph cards if encountering them during card gathering. (It is still not possible for Document to include the contents of cards having user-defined substance types such as NCFile cards.)

Data saved at card closing:

When a card is closed, only those parts that are dirty are written out to the notefile. A message indicating which parts are being saved is now printed to the card's prompt window during closing. Furthermore, certain card types (in particular, browsers) were saving their substance even if no changes were made. This source of space inefficiency has been fixed in Release1.2i.

Ordering cards in a filebox:

It is now possible to dictate the relative placement of new cards in a filebox. If the OrderingFn property of a card has a value, it should be a lisp function that takes two card ID arguments and returns T if the first should appear before the second and NIL otherwise. You can make such a function appear automatically on new boxes for the case of alphabetizing by using the global parameter AlphabetizeFileBoxChildren. See section 2.

Programmer's interface:

The Programmer’s interface has been updated. Thus users with existing programmer’s interface code should read the revised PI documentation. The changes are not all forward compatible.

Notecards system date:

You can find the date of your Notecards system in the variable NC.SystemDate. The ’NewestFile property on the NC.SystemDate atom contains the name of the last modified Notecards file.

The Notecards library packages:

The old Release1.1 library packages have been converted to 1.2i and documented and several new ones have been added. These can be found on {qv}<notecards>release1.2>library> and include NCScreen, NCCluster, NCChain, NCFileCard, NCKeys, NCHacks, and ARIDemo. Documentation can be found in <filename>.ted.

NCScreen defines several handy functions for arranging cards on the screen callable from the programmer’s interface. NCCluster defines several new card types, most notably CaseCluster, a cluster of cards for use in the sample domain of legal case analysis. NCChain defines the Chain card type, useful for breaking up a large text card into a linked chain of cards. NCFileCard defines the new File card type and FILE substance allowing a notefile to link to external files via standard Notecards links. NCKeys provides a shorthand language for invoking various handy programmer’s interface functions. NCHacks contains several handy functions written using the programmer’s interface. Two of these allow global text searches and replaces throughout a notefile. In addition there is a function that searches by last card modification date and one that links cards to form chains. Finally, ARIDemo is an example of how the programmer’s interface can be used to construct notefiles that demo themselves.

Loading NoteCards from different directories:

NoteCards now uses the values of four directories variables to decide from whence to load the code. These are NOTECARDSDIRECTORIES, NOTECARDSMAPDIRECTORIES, QUADTREEDIRECTORY, and MAPFILEDIRECTORY. They default to ({QV}<NOTECARDS>RELEASE1.2I>), ({QV}<NOTECARDS>MAPS>NEW>), {QV}<NOTECARDS>MAPS>, and {QV}<NOTECARDS>MAPS> respectively.

5. Known bugs and plans for future improvements:

o The compactor should check first for available space.

o There are major speed problems in redrawing large browsers. Changing link display mode could also use some streamlining.

o Integrate the document compiler and the types mechanism so that instances of new card types can be sucked into TEdit documents.

o Make links into full-fledged objects having properties and type hierarchies.

The NoteCards Types Mechanism

Release 1.2

Frank G. Halasz

Xerox PARC

First Written: 22-Mar-85

Modified: 26-Mar-85 by Frank G. Halasz

Modified: 1-Aug-85 by Lissa Monty

1. Introduction

The NoteCards types mechanism allows a user with some knowledge of Interlisp to add new types of note cards to the system. The types mechanism is built around an inheritance hierarchy of note card types. If the user needs to create a new card type that is a small change from an already existing card type, he or she need only define the few functions or parameters that account for the differences between the new card and the existing card. However, if the user wishes to create a totally new type of card, then he or she must define the 20-odd functions and parameters that make up a note card type.

Every note card has a substance. A substance is essentially a data structure that contains the information in the note card. Different types of note cards have different types of substances. Associated with every substance type is an editor that can be used to create and/or modify the data structure of that substance type. For example, the substance of a Text card is a TEXTSTREAM that can be edited using TEdit. Similarly, the substance of a Browser card is a GRAPH record that can be edited using GRAPHER. Defining a new note card type involves specifying the functions necessary to handle the card's substance and its editor.

1.1 The Inheritance Hierarchy

The inheritance hierarchy in NoteCards has two parts: a tree of NoteCardTypes and a list of SubstanceTypes. Every NoteCardType has a super-type and a substance type. The super-type is an already existing NoteCardType from which the NoteCardType will inherit fields. Thus, the set of NotecardTypes forms a tree structure based on the super-type field. The substance type of a NotecardType is an already existing SubstanceType.

The inheritance process for a given field of a NoteCardType works as follows: if the field has a non-NIL value in the NoteCardType then this value is used, otherwise the field value is inherited from its super-type. If there are no non-NIL values anywhere in the inheritance path for the NoteCardType, then the field value is taken from the corresponding field in the substance type for the NoteCardType. Substance types are guaranteed to have values in all of their fields.

Example: ProtectedText is a card with super-type Text. Text in turn has super-type NoteCard (the null root of the NoteCardType tree). In addition, Text has substance type TEXT. If an EditCardFn is not defined in ProtectedText, then it will be inherited from Text. If Text doesn't have an EditCardFn then the EditSubstanceFn from the TEXT SubstanceType will be used (since NoteCard by definition does not have an EditCardFn).

Functions are inherited all or none. Often, however, a new NoteCardType will require only a minor addition to the corresponding function of its super-type. In this case, the new card type should define a new function, but this function can call the corresponding function of its super-type to do the bulk of the work. The following construction will accomplish this goal:

(APPLY* (NCP.CardTypeInheritedField (NCP.CardTypeSuper *<type>*) *<fn>*) *<arg1>* *<arg2>* ...)

where *<type>* is the TypeName of the card type in question, *<fn>* is the name of the function in question, and *<arg1>* *<arg2>* ... are the arguments to that function. For example the following might be the definition of the EditCardFn for the passworded Text card called ProtectedText:

```
(DEFINEQ

(NC.EditProtectedTextCard
  (LAMBDA (ID Substance Region/Position)
          (* * Edit a Protected Text card, asking for the password first.)
    (PROG (Password Result)
            (* * Get this card's password from the prop list)
          (SETQ Password (NCP.CardProp ID (QUOTE Password)))
          (COND
            ((EQUAL Password (NC.GetPassword ID))
              (* Password is okay.
                 Call the EditCardFn of my super-type)
              (SETQ Result (APPLY*
                   (NCP.CardTypeInheritedField
                     (NCP.CardTypeSuper (QUOTE ProtectedText))
                     (QUOTE EditCardFn))
                   ID Substance Region/Position)))
            (T (* Password is bad. Express condolences)
               (NCP.PrintMsg Window T "Sorry." (CHARACTER 13)
                             "You do not know the password!!"
                             (CHARACTER 13)
                             "Bye."
                             (CHARACTER 13))
               (DISMISS 2000)))
          (RETURN Result))))
```

1.2 Links and Link Icons

An integral part of NoteCards is the ability to create a link between two note cards. Presently, there are two kinds of links: *GlobalToGlobal* links and *LocalToGlobal* links. GlobalToGlobal links connect one entire card with another entire card and are stored separately from either card's substance.

GlobalToGlobal links are maintained (almost) entirely by the NoteCards system code and therefore do not vary across note card types.

LocalToGlobal links connect a particular position within the substance of one card (the *source* card) to the entirety of the other card (the *destination* card). Within the source card, the link is represented by an image object called a link icon that must be contained by the card's substance. Since substances vary across note card types, the handling of link icons varies across note card types. The destination (or Global) end of a LocalToGlobal links is maintained by the NoteCards system code.

Not all note cards can be the source of LocalToGlobal links. Card types that support LocalToGlobalLinks must have their LinkAnchorModesSupported parameter set to T. If a this parameter has any other value, then cards of this type can be the source of only GlobalToGlobal links. These Global-links-only card types need to provide only one piece of functionality in support of the linking mechanism. In particular, they must provide user access to the function **NCP.GlobalGlobalLink** from the editor that runs when the card is being displayed. For example, the editor's command menu might include an "Insert Global Link" command. All other link maintenance is carried out by the NoteCards system.

If a card type supports LocalToGlobal links, then it must contain the necessary mechanisms for supporting link icons in its substance. Link icons are instances of standard Interlisp-D image objects (See documentation of Image Objects in Interlisp-D). The mechanisms supporting link icons include functions for inserting, deleteing, updating, and collecting the link icons contained in a card's substance. These functions are described in detail below. In addition to these functions, a note card type supporting LocalToGlobal links must provide user access to the function **NCP.LocalGlobalLink** from the editor that runs when the card is being displayed. In addition the editor must provide user access to the function **NCP.GlobalGlobalLink**.

Inside the link icon image object is a link record containing all of the information about the link. These link records can be manipulated using the link manipulation functions provided by NoteCards' programmer's interface (e.g., **NCP.GetLinkDestination** returns the destination field of a link record). The functions required to define a note card or substance type deal in both link records and link icons. You can translate between these two representations using the functions **NC.MakeLinkIcon** and **NC.FetchLinkFromLinkIcon**; **NC.MakeLinkIcon** will create a link icon image object from a link record, while **NC.FetchLinkFromLinkIcon** will return the link record contained in a link icon.

1.3 Using the Types Mechanism

Most uses of the types mechanism involve defining new NoteCardTypes. Usually, these new NoteCardTypes involve specifying a TypeName, a SuperType, a SubstanceType, and one or two functions that differ from the SuperType. The most commonly defined functions are the MakeCardFn, the EditCardFn and the QuitCardFn.

Definition of new substance types occurs only when a new kind of substance (e.g., a spreadsheet) and its corresponding editor are to be added to the system. When defining a substance, all of its fields must be fully defined since there is no inheritance among SubstanceTypes.

2. The NoteCardType

Each note card type in the system is defined by a record structure (i.e., a NoteCardType) containing about 20 names, functions and parameters. The functions implement behaviors that are required by the NoteCards system but vary across the different card types. For example, one function is responsible for writing the card's substance to the NoteFile. The parameters represent specifications that inform NoteCards about the specific properties of each card type, e.g., whether it handles local links or not.

The NoteCardTypes are organized into an inheritance hierarchy. Each NoteCardType has a super-type. If any of the functions or parameters is not specified for a given NoteCardType, that function or parameter is inherited from its super-type (or its super-type's super-type, if the function or parameter is not specified for the super-type either). Each NoteCardType also has a SubstanceType. If any of the functions or parameters cannot be found along the super-type chain of the NoteCardType, then the card type inherits the function or parameter from its SubstanceType.

Overall, a card type is a data structure with the following 21 fields:

Inheritance Hierarchy Specifications

1) TypeName
2) SuperType
3) SubstanceType

Functions

4) MakeCardFn
5) EditCardFn
6) QuitCardFn
7) GetCardFn
8) PutCardFn
9) CopyCardFn
10) MarkCardDirtyFn
11) CardDirtyPFn
12) CollectLinksInCardFn
13) DeleteLinksInCardFn
14) UpdateLinkIconsInCardFn
15) InsertLinkInCardFn
16) TranslateWindowPositionToCardPositionFn

Parameters

17) LinkDisplayMode
18) CardDefaultWidth
19) CardDefaultHeight
20) CardLinkAnchorModesSupported
21) CardDisplayedInMenuFlg

These fields are defined as follows:

1. TypeName: The atom that is the name of this card type. TypeNames must be unique among the NoteCardTypes tree, though they may overlap with SubstanceNames. The convention is that NoteCardType TypeNames have only the first letter capitalized. This is to set them apart from SubstanceNames which are by convention all caps.

2. SuperType: The TypeName of the NoteCardType that is the super-type for this NoteCardType. When a new NoteCardType is created, its SuperType must be an existing NoteCardType.

3. SubstanceType: The SubstanceName for the substance of this card type. When a new card is created, its SubstanceType must be the name of an existing SubstanceType (see Section 3.0 below). The basic NoteCards system includes the following substance types: TEXT, SKETCH, GRAPH which represent the the substances handled by the TEdit, Sketch, and Grapher packages respectively.

4. MakeCardFn: The name of a function to be applied to an ID, a Title, and a NoDisplayFlg. The function should create a new card of this type. The ID is the note card ID that will be assigned to the newly created card. It should be used to set the various properties of the new card. The title is a string specifying the title of the new card. It can be used in messages to the user or to set the title of any windows created. NoDisplayFlg determines whether the new card is to be displayed on the screen or not. If NoDisplayFlg is non-NIL, then the card is to be displayed in a window on the screen. If NoDisplayFlg is NIL, then the card is to be created but not displayed on the screen.

The MakeCardFn should return the window of the new card if NoDisplayFlg is non-NIL and the ID if NoDisplayFlg is NIL.

Before returning, every MakeCardFn is required to set the substance property of ID by calling (**NC.SetSubstance** ID *Substance*) where *Substance* is whatever is considered a substance for this card type. For example, a TextStream for Text cards, a Graph record for Graph cards, or a Sketch record for Sketch cards.

By convention, every MakeCardFn sets the SHRINKFN of any window it creates to the function **NC.ShrinkFn** using WINDOWPROP.

5. EditCardFn: The name of a function to be applied to ID, Substance, and Region/Position. The function should start an editor for the given card. ID is the note card ID of the card. Substance is the substance of the card; it will be a thing of whatever type is considered a substance for this card type, e.g., a TextStream or Sketch record. Region/Position is a Region or a Position on the screen that specifies where the card is to be placed. (**NC.DetermineDisplayRegion** ID Region/Position) is a function that will determine the exact region for the card’s window given the ID and the Region/Position.

The EditCardFn should return the editor window.

The EditCardFn is responsible for checking to make there is not already an editor for card ID already on the screen. If there is, the EditCardFn should just flash the previous editor window.

By convention an EditCardFn sets the SHRINKFN of any window it creates to the function **NC.ShrinkFn** using WINDOWPROP. Also by convention, an EditCardFn should set the title of the editor window to be the value of (**NCP.CardTitle** ID).

6. QuitCardFn: The name of a function to be applied to WindowOrSubstanceOrID which is either the editor window for a card or the substance of a card or a note card ID. QuitCardFn should quit out of the editor currently operative on the specified card and close the window containing the card.

The value returned by QuitCardFn is unspecified.

Before returning the QuitCardFn should apply the function **NC.DeactivateCard** to the ID of the card. Note that the ID may have to be computed from the Window or Substance passed to the QuitCardFn. The function **NC.CoerceToID** will do this computation.

The QuitCardFn should also insure that all processes related to this card are completed (or guaranteed to eventually complete) before returning.

7. GetCardFn: The name of a function to be applied to the DatabaseStream, a card ID, and a screen Region. The GetCardFn should read the substance of the note card specified by ID from the DatabaseStream. The format of the data to be read is determined by the PutCardFn (see below). When the GetCardFn is called, the file pointer for DatabaseStream is positioned on the first byte of the data to be read.

The GetCardFn should return a pointer to the substance read from the DatabaseStream.

GetCardFn need produce no side-effects. The ID and the Region are for reference purposes only.

Note that the GetCardFn need only read the substance of the card, i.e., that information about the card which is specific to its card type. General information about a card such as its title, its property list, its list of links, etc. is read from the DatabaseStream by the system.

8. PutCardFn: The name of a function to be applied to a note card ID and the DatabaseStream. The PutCardFn should write the substance of the note card specified by ID to the DatabaseStream. When the PutCardFn is called, the file pointer for DatabaseStream is positioned at the first byte assigned to the card. When the PutCardFn returns, the file pointer should be positioned immediately after the last byte written.

The format for writing the card's substance is fairly unrestricted. The data written on the DatabaseStream can take up any number of bytes, but the bytes must be contiguous. It must be written so that it can recovered by reading from the DatabaseStream using the GetCardFn. The only other restriction is that the first 6 bytes of the substance must contain the file position of the start and the end of the substance: 3 bytes for the start file pointer and 3 bytes for the end file pointer. These pointers are for use by the CopyCardFn.

The value returned by the PutCardFn is unspecified.

Note that the PutCardFn need only write out the substance of the card, i.e., that information about the card which is specific to its card type. General information about a card such as its title, its property list, its list of links, etc. is written to the DatabaseStream by the system.

9. CopyCardFn: The name of a function to be applied to a note card ID, a "from" DatabaseStream, and a "to" DatabaseStream. The CopyCardFn should copy the substance for the note card specified by ID from the "from" DatabaseStream to the "to" DatabaseStream. When the CopyCardFn is called the file pointer for the "from" DatabaseStream is positioned on the first byte of the data to be copied. The file pointer for the "to" DatabaseStream is positioned at the first byte of the space assigned to the card on the "to" DatabaseStream.

The format for writing the substance on the "to" DatabaseStream has the same restrictions as for the PutCardFn.

Most often the the CopyCardFn is a simple COPYBYTES that uses the start and end pointers written by PutCardFn in the first 6 bytes of the substance. Note, however, that all file absolute pointers (including the start and end pointers) must be updated; the file location on the "to" DatabaseStream is almost never the same as the original file location on the "from" DatabaseStream.

The value returned by the CopyCardFn is unspecified.

The CopyCardFn is used primarily by the compactor that eliminates "dead" space in the database. Thus, it is important that the CopyCardFn be as time efficient as possible.

10. MarkCardDirtyFn: The name of a function to be applied to a note card ID and a ResetFlg. If the ResetFlg is non-NIL, the function should mark the card specified by ID as being dirty (i.e., changed since it was last written to the DatabaseStream). If the ResetFlg is NIL, the function should reset the "dirtiness" of the card.

The MarkCardDirtyFn is called by NoteCards system functions that change the card. It is not necessarily called by user operations inside the editor on the card. Therefore, it is best if the mechanism used by the MarkCardDirtyFn is somehow coordinated with the corresponding mechanism used by the editor on the card. (See the CardDirtyPFn below.)

The value returned by the MarkCardDirtyFn is unspecified.

The card specified by ID is guaranteed to be active.

11. CardDirtyPFn: The name of a function to be applied to a note card ID. The function should return a non-NIL value if the card specified by ID is dirty, i.e., if it was changed since it was last written to the DatabaseStream. NIL should be returned otherwise.

Note that a "dirty" card is one that has been changed in any way. Only NoteCards specific changes to a card will result in a call to the card's MarkCardDirtyFn. Changes made through the editor on the card will use the editors "mark dirty" mechanism and will not call the MarkCardDirtyFn. Therefore, the CardDirtyPFn should check all dirty flags, i.e., the dirty flag set by the MarkCardDirtyFn as well as any set by the card's editor.

The card specified by ID is guaranteed to be active.

12. CollectLinksInCardFn: The name of a function to be applied to a note card ID, a CheckAndDeleteFlg, a DatabaseStream, a ReturnLinkIconsFlg, and a ReturnLocationsFlg. The function should examine the substance of the card specified by ID and produce a list of the links (or link icons) contained by the substance. The ReturnLinkIconsFlg and the ReturnLocationsFlg determine the contents of the list to be returned as follows:

> ReturnLinkIconsFlg and ReturnLocationsFlg both NIL: the list to be returned should be a list of link records.
>
> ReturnLinkIconsFlg is non-NIL, ReturnLocationsFlg is NIL: the list to be returned should be a list of link icons.
>
> ReturnLinkIconsFlg is NIL, ReturnLocationsFlg is non-NIL: the list to be returned should be a list of pairs where the first memeber of the pair is a link record and the second member of the pair is the "location" of the link icon for that link inside the substance.
>
> ReturnLinkIconsFlg and ReturnLocationsFlg both non-NIL: the list to be returned should be a list of pairs where the first memeber of the pair is a link icon and the second member of the pair is the "location" of that link icon.

If CheckAndDeleteFlg is non-NIL, then the list produced by CollectLinksInCardFn should contain valid links only. Any links found to be invalid should be deleted. To check the validity of a link, the function **NC.ValidLinkP** should be applied to the link record and the DatabaseStream. To delete a link, apply the function **NC.MakeInvalidLink** to the link icon.

The CollectLinksInCardFn should return the list produced CONSed to a dirty flag. The dirty flag should be non-NIL if any links were deleted, NIL otherwise.

The card specified by ID is guaranteed to be active.

13. DeleteLinksInCardFn: The name of a function to be applied to a "source" note card ID and a link record or "destination" note card ID. If the second argument is a link, the function should remove from the substance of the card specified by "source" ID the link icon corresponding to link. If the second argument is a "destination" note card ID, the function should remove from the substance of the card specified by "source" ID all link icons corresponding to links pointing to the card specififed by "destination" ID.

To "remove" a link icon, the link icon should be replaced in the substance by the image object that is the value of **NC.DeletedLinkImageObject**. Note that before deleting the link icon, it is best to replace the IMAGEOBJFNS of the link icon with the value of **NC.NoDeleteImageFns**. This will prevent the link icon's WHENDELTEDFN from being activated when the deletion takes place.

The value returned by the DeleteLinksInCardFn is unspecified.

The card specified by "source" ID is guaranteed to be active.

14. UpdateLinkIconsInCardFn: The name of a function to be applied to a "source" note card ID or window and a "destination" note card ID. The function should update (i.e., force a redisplay of) all link icons in the "source" card that represent links pointing to the "destination" card. This function is called when some property of the link is changed by the NoteCards code. It is also called when certain properties of the destination card (e.g., its title) are changed.

The value returned by the UpdateLinkIconsInCardFn is unspecified.

The "source" card is guaranteed to be active.

15. InsertLinkInCardFn: The name of a function to be applied to a window, a link, and a position. The function should insert a link icon containing the link into the card being edited in the window at the position specified. The position is whatever object is returned by the TranslateWindowPositionToCardPositionFn.

The value returned by the InsertLinkInCardFn is unspecified.

The ID of the card being edited by the window is guaranteed to be the SOURCEID of the link.

16. TranslateWindowPositionToCardPositionFn: The name of a function to be applied to a window, an X-coordinate in that window, and a Y-coordinate in that window. The window is an editor window on the substance of some card. The function should return a position object that describes the position in the card substance that is currently located at the given X-Y position in the window. The format of the position object is undefined. It will be passed to the InsertLinkInCardFn and used as the position at which to insert a links in the card being edited in the window.

17. LinkDisplayMode: determines the default display mode for link icons inserted into cards of this type. It must be a record of type LINKDISPLAYMODE. LINKDISPLAYMODE describes what information will be displayed in a link icon. It consists of three flags: SHOWTITLEFLG, SHOWLINKTYPEFLG, and ATTACHBITMAPFLG. If SHOWTITLEFLG is non-NIL, the link icon will display the destination card's title. If SHOWLINKTYPEFLG is non-NIL, the link icon will display the type of the link. If ATTACHBITMAPFLG is non-NIL, a bit map describing the type of the destination card will be attached to the right of the link icon.

Note: This property in NOT inherited.

18. CardDefaultWidth: The default width for editor windows on cards of this type.

19. CardDefaultHeight: The default height for editor windows on cards of this type.

20. CardLinkAnchorModesSupported: an atom that determines the kind of links this card type will support (i.e., the kind of links for which cards of this type can be a source). If NIL, then this card type does not support links of any type. If Global, this card supports only Global links. If Local, this card supports only local links. If T, this card supports both Global and Local links.

Note: This property in NOT inherited.

21. CardDisplayedInMenuFlg: if non-NIL then this card type will appear in the choice of card types in the menu used during card creation using the "Create" entry in the main NoteCards menu. If NIL, then this card type will not appear in this menu.

3. The SubstanceType

The SubstanceType is a record structure whose fields are virtually identical to those of the NoteCardType record. In particular, the SubstanceType has the following 17 fields:

1) SubstanceName

2) CreateSubstanceFn

3) EditSubstanceFn

4) QuitSubstanceFn

5) GetSubstanceFn

6) PutSubstanceFn

7) CopySubstanceFn

8) MarkSubstanceDirtyFn

9) SubstanceDirtyPFn

10) CollectLinksInSubstanceFn

11) DeleteLinksInSubstanceFn

12) UpdateLinkIconsInSubstanceFn

13) InsertLinkInSubstanceFn

14) TranslateWindowPositionToSubstancePositionFn

15) SubstanceDefaultWidth

16) SubstanceDefaultHeight

17) SubstanceLinkAnchorModesSupported

These fields are defined as follows:

1. SubstanceName: The atom that is the name of this substance type. SubstanceNames must be unique among the substance types, though they may overlap with card TypeNames. The convention is that SubstanceNames are all in caps. This is to set them apart from card TypeNames which by convention have only their first letter capitalized.

2 Thru 14. Functions: All of the functions are identical to the corresponding functions in the NoteCardType record structure. Note the (arbitrary) use of "create" instead of "make" in the name of the CreateSubstanceFn.

15 Thru 17. Parameters: The parameters are identical to the corresponding parameters in the NoteCardType data structure. There are no parameters for the LinkDisplayMode and the DisplayInMenuFlg because these two parameters are not inherited. They must be specified separately for each card type.

4. Adding a New NoteCardType or SubstanceType to the System

The functions **NCP.CreateCardType** and **NCP.CreateSubstanceType** can be used to add new Types to the system.

NCP.CreateCardType takes 5 arguments: the TypeName, its SuperType, its SubstanceType, a functions list, and a parameters list. The functions list is an ASSOC list where the CAR of each sub-list is one of the function field names given above (e.g., EditCardFn, MakeCardFn, etc.). The CDR of the sublist should contain the name of the required function. Any function field name for which there is no entry will be set to NIL and will thus be inherited. The parameters list is analogous to the functions list, except that it applies to the parameter field names (i.e., LinkDisplayMode, CardDefaultWidth, CardDefaultHeight, and CardLinkAnchorModesSupported).

NC.CreateSubstanceType takes 3 arguments: the SubstanceName, a functions list, and a parameters list. The functions and parameters list are analogous to those for **NCP.CreateCardType** except that all of the function and parameter fields specified above MUST have an entry in the ASSOC lists.

Both **NCP.CreateCardType** and **NC.CreateSubstanceType** will overwrite existing types (NoteCard and Substance, respectively) of the same name.

5. Example: Defining the ProtectedText NoteCardType

The following is an example of defining a new card type called the ProtectedText card. The card type is created by specifying new MakeCardFn and EditCardFn functions. All other functions are inherited from from the super-type, i.e., the Text card. All of the parameters are specified directly for this card.

· The function that creates the new ProtectedText card type:

```
(NC.AddProtectedTextCardType
  (LAMBDA NIL                        (* fgh: "26-Mar-85 15:48")
          (* * Create the ProtectedText card type)
    (NCP.CreateCardType (QUOTE ProtectedText)
                        (QUOTE Text)
                        (QUOTE TEXT)
                        (QUOTE ((MakeCardFn NC.MakeProtectedTextCard)
                                (EditCardFn NC.EditProtectedTextCard)))
                        (QUOTE ((LinkDisplayMode (T NIL NIL))
                                 (CardDefaultHeight 300)
                                 (CardDefaultWidth 400)
                                 (CardLinkAnchorModesSupported T)
                                 (CardDisplayInMenuFlg T))))))
```

· The MakeCardFn for the ProtectedText card type:

```
(NC.MakeProtectedTextCard
  (LAMBDA (ID Title NoDisplayFlg)    (* fgh: "26-Mar-85 15:23")
          (* * Make a protected Text card
               by calling the make card fn for a Text card
               and then attaching a password to the card)
    (PROG (Window WindowOrID)
          (* * Create the Text card)
          (SETQ WindowOrID (APPLY*
              (NCP.CardTypeFn (NCP.CardTypeSuper
                                 (QUOTE ProtectedText))
                               (QUOTE MakeCardFn))
              ID Title NoDisplayFlg))
          (* * Get the window for the card, if there is one)
          (SETQ Window (WINDOWP WindowOrID))
          (* * Get the password from the user
               and add it to the cards prop list)
          (NCP.CardProp ID (QUOTE Password)
                        (NC.GetPassword Window))
          (* * Return whatever the super-type's MakeCardFn returned)
          (RETURN WindowOrID))))
```

· The EditCardFn for the ProtectedText card type:

```
(NC.EditProtectedTextCard
  (LAMBDA (ID Substance Region/Position)
                                    (* fgh: "26-Mar-85 17:21")
          (* * Edit a Protected Text card, asking for the password first.)
    (PROG (ExactRegion Window Password Result)
          (* * Open a window for this card)
          (SETQ ExactRegion (NC.DetermineDisplayRegion ID
                                         Region/Position))
          (SETQ Window (CREATEW ExactRegion))
          (* * Get this card's password from the prop list)
          (SETQ Password (NCP.CardProp ID (QUOTE Password)))
          (COND
            ((EQUAL Password (NC.GetPassword Window))
              (* Password is okay.
                 Call the EditCardFn of my super-type)
              (SETQ Result (APPLY*
                  (NCP.CardTypeInheritedField
                    (NCP.CardTypeSuper (QUOTE ProtectedText))
                    (QUOTE EditCardFn))
                  ID Substance ExactRegion)))
            (T (* Password is bad. Express condolences)
               (NCP.PrintMsg Window T "Sorry." (CHARACTER
                               13)
                             "You do not know the password!!"
                             (CHARACTER 13)
                             "Bye."
                             (CHARACTER 13))
               (DISMISS 2000)))
          (* * Close the window you created.
               The super-types EdityCardFn will
               have created another window.)
          (CLOSEW Window)
          (RETURN Result))))
```

· A utility used by the MakeCardFn and the EditCardFn:

```
(NC.GetPassword
  (LAMBDA (Window)                  (* fgh: "26-Mar-85 15:50")
          (* * Get a password from the user.
               Window is the main window for the card in question)
    (NCP.AskUser "What is the password for this card?" " -- "
               NIL T Window)))
```

Venue

Medley for the Novice

Release 2.0
February, 1992

Address comments to:
Venue
User Documentation
1549 Industrial Road
San Carlos, CA 94070
415-508-9672

Medley for the Novice

Release 2.0

February 1992

Text was written and produced with Venue text formatting tools; Xerox printers were used to produce text masters. The typeface is Classic.

1. BRIEF GLOSSARY

The following definitions will acquaint you with general terms used throughout this primer. You will probably want to read through them now, and use this chapter as a reference while you read through the rest of the primer.

advising	A Medley facility for specifying function modifications without necessarily knowing how a particular function works or even what it does. Even system functions can be changed with advising.
argument	A piece of information given to a Lisp function so that it can execute successfully. When a function is explained in the primer, the arguments that it requires will also be given. Arguments are also called Parameters.
atom	The smallest structure in Lisp; like a variable in other programming languages, but can also have a property list and a function definition.
Background Menu	The menu that appears when the mouse is not in any window and the right mouse button is pressed.
binding	The value of a variable. It could be either a local or a global variable. See unbound.
bitmap	A rectangular array of "pixels," each of which is on or off representing one point in the bitmap image.
`BREAK`	An Lisp function that causes a function to stop executing, open a Break window, and allows you to find out what is happening while the function is halted.
Break Window	A window that opens when an error is encountered while running your program (i.e., when your program has broken). There are tools to help you debug your program from this window. This is explained further in Chapter 14.
browse	To examine a data structure by use of a display that allows you to "move" around within the data structure.
button	(1) (n.) A key on a mouse. (2) (v.t.) To press one of the mouse keys when making a selection.
`CAR`	A function that returns the head or first element of a list. See `CDR`.
caret	The small blinking arrowhead that marks where text will appear when it is typed in from the keyboard.
`CDR`	A function that returns the tail (that is, everything but the first element) of a list. See `CAR`.

CLlSP A mechanism for augmenting the standard Lisp syntax. One such augmentation included in Interlisp is the iterative statement. See Chapter 9.

cr Press your Return key.

datatype (1) The kind of a datum. In Interlisp, there are many system-defined datatypes, e.g., Floating-Point, Integer, Atom, etc.

(2) A datatype can also be user-defined. In this case, it is like a record made up from system types and other user-defined datatypes.

DWIM "Do-what-I-mean." Many errors made by Medley users could be corrected without any information about the purpose of the program or expression in question (e.g., misspellings, certain kinds of parenthesis errors). The DWIM facility is called automatically whenever an error occurs in the evaluation of an Interlisp expression. If DWIM is able to make a correction, the computation continues as though no error had occurred; otherwise, the standard error mechanism is invoked.

error Occasionally, while a program is running, an error may occur which will stop the computation. Interlisp provides extensive facilities for detecting and handling error conditions, to enable the testing, debugging, and revising of imperfect programs.

evaluate or EVAL To find the value of a form. For example, if the variable X is bound to 5, we get 5 by evaluating X. Evaluation of a Lisp function involves evaluating the arguments and then applying the function.

Executive Window This is your main window, where you will run functions and develop your programs. This is the window that the caret is in when you turn on your machine and load Medley.

file package A set of functions and conventions that facilitate the bookkeeping involved with working in a large system consisting of many source code files and their compiled counterparts. Essentially, the file package keeps track of where things are and what things have changed. It also keeps track of which files have been modified and need to be updated and recompiled.

form Another way of saying s-expression. A Lisp expression that can be evaluated.

function A piece of Lisp code that executes and returns a value.

history The programmer's assistant is built around a memory structure called the history list. The history functions (e.g. FIX, UNDO, REDO) are part of this assistant. These operations allow you to conveniently rework previously specified operations.

History List As you type on the screen, you will notice a number followed by a slash, followed by another number. The first number is the exec number, the second is the event number. Each number, and the information on that line, is stored sequentially as the History List Using the History List, you can easily reexecute lines typed earlier in a work session. See Chapter 2.

icon — A pictorial representation, usually of a shrunken window.

inspector — An interactive display program for examining and changing the parts of a data structure. Medley has inspectors for lists and other data types.

iterative statement — (also called i.s.) A statement in Interlisp that repetitively executes a body of code For example, `(for x from 1 to 5 do (PR1NT x))` is an i.s.

iterative variable — (also called i.v.) Usually, an iterative statement is controlled by the value that the i.v. takes on. In the iterative statement example above, x is the iterative variable because its value is being changed by each cycle through the loop. All iterative variables are local to the iterative statement where they are defined.

Lisp — Family of languages invented for "list processing." These languages have in common a set of basic primitives for creating and manipulating symbol structures. Interlisp-D is an implementation of the Lisp language together with an environment (set of tools) for programming, and a set of packages that extend the functionality of the system.

list — A collection of atoms and lists; a list is denoted by surrounding its contents with a pair of parentheses.

Masterscope — A program analysis tool. When told to analyze a program, Masterscope creates a database of information about the program. In particular, Masterscope knows which functions call other functions and which functions use which variables. Masterscope can then answer questions about the program and display the information with a browser.

menu — A way of graphically presenting you with a set of options. There are two kinds of menus: pop-up menus are created when needed and disappear after an item has been selected; permanent menus remain on the screen after use until deliberately closed.

mouse — The mouse is the box attached to your keyboard. It controls the movement of the cursor on your screen. As you become familiar with the mouse, you will find it much quicker to use the mouse than the keyboard.

Mouse Cursor — The small arrow on the screen that points to the northwest.

Mouse Cursor Icons — Four types of mouse cursor icons are shown below.

Wait. The processor is busy.

The Mouse Confirm Cursor. It appears when you have to confirm that the choice you just made was correct. If it was, press the left button. If the choice was not correct, press the right button to abort.

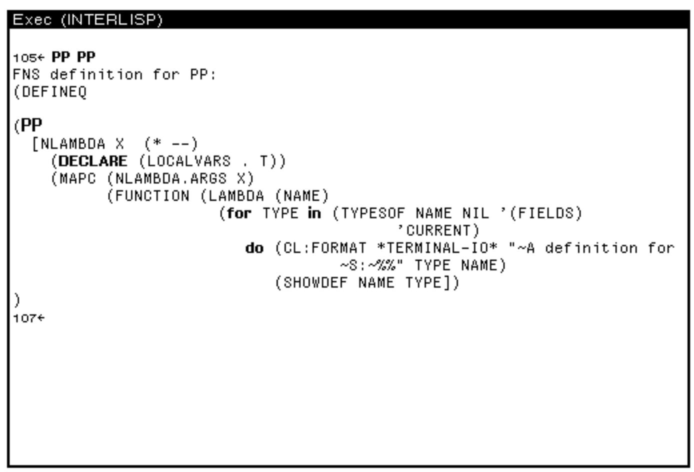
This means "sweep out" the shape of the window. To do this, move the mouse to a position where you want a corner. Press the left mouse button, and hold it down. Move the mouse diagonally to sketch a rectangle. When the rectangle is the desired size and shape, release the left button.

This is the "move window" prompt. Move the mouse so that the large "ghost" rectangle is in the position where you want the window. When you click the left mouse button, the window will appear at this new location.

`NIL` — `NIL` is the Lisp symbol for the empty list. It can also be represented by a left parenthesis followed by a right parenthesis ( ). It is the only expression in Lisp that is both an atom and a list.

pixel — Pixel stands for "picture element." The computer monitor screen is made up of a rectangular array of pixels. Each pixel corresponds to one bit. When a bit is turned on (i.e., set to 1), the pixel on the screen represented by this bit is black.

pretty printing — Pretty printing refers to the way Lisp functions are printed with special indentation, to make them easier to read. Functions are pretty printed in the structure editor, SEdit (see Chapter 7). You can pretty print uncompiled functions by calling the function `PP` with the function you would like to see as an argument, i.e. (`PP` function-name). For an example of this, see Figure 1.5.



```
Exec (INTERLISP)

105← PP PP
FNS definition for PP:
(DEFINEQ

(PP
  [NLAMBDA X  (* --)
    (DECLARE (LOCALVARS . T))
    (MAPC (NLAMBDA.ARGS X)
          (FUNCTION (LAMBDA (NAME)
                      (for TYPE in (TYPESOF NAME NIL '(FIELDS)
                                           'CURRENT)
                         do (CL:FORMAT *TERMINAL-IO* "~A definition for
                                ~S:~%%" TYPE NAME)
                            (SHOWDEF NAME TYPE])
)
107←
```

Figure 1.5. Example of Pretty Printing Function `PP`

Programmer's Assistant — The programmer's assistant accesses the History List to allow you to `FIX`, `UNDO`, and/or `REDO` your previous expressions typed to the executive window (see Chapter 2).

Prompt Window — The narrow black window at the top of the screen. It displays system prompts, or prompts you have developed (see Figure 1.6).

Figure 1.6. Prompt Window

property list A list of the form (<property-namel> <property-value1> <property-name2> <property-value2> ....) associated with an atom. It accessed by the functions GETPROP and PUTPROP.

record A record is a data structure that consists of named "fields". Accessing elements of a record can be separated from the details of how the data structure is actually stored. This eliminates many programming details. A record definition establishes a record template, describing the form of a record. A record instance is an actual record storing data according to a particular record template. (See datatype, second definition.)

Right Button Default Window Menu This is the menu that appears when the mouse is in a window, and the right mouse button is pressed. It looks like the menu in Figure 1.7. If this menu does not appear when you press the right button of the mouse and the mouse is in the window, move the mouse so that it is pointing to the title bar of the window, and press the right button.

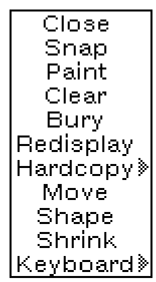


Figure 1.7. Right Button Default Window Menu

s-expression Short for "symbolic expression". In Lisp, this refers to any well-formed collection of left parentheses, atoms, and right parentheses.

stack A pushdown list. Whenever a function is entered, information about that specific function call is pushed onto (i.e., added to the front of) the stack. This information includes the variable names and their values associated with the function call. When the function is exitted, that data is popped off the stack.

sysout A flle containing a whole Lisp environment: namely, everything you defined or loaded into the environment, the windows that appeared on the screen, the amount of memory used, and so on. Everything is stored in the sysout file exactly as it was when the function SYSOUT was called.

`TRACE` A function that creates a trace of the execution of another function. Each time the traced function is called, it prints out the values of the arguments it was called with, and prints out the value it returns upon completion.

unbound Without value; an atom is unbound if a value has never been assigned to it.

window A rectangular area of the screen that acts as the main display area for some Lisp process,

PREFACE

It was dawn and the local told him it was down the road a piece, left at the first fishing bridge in the country, right at the appletree stump, and onto the dirt road just before the hill. At midnight he knew he was lost. -Anonymous

Welcome to the Medley Lisp Development Environment, a collection of powerful tools for assisting you in programming in Lisp, developing sophisticated user interfaces, and creating prototypes of your ideas in a quick and easy manner. Unfortunately, along with the power comes mind-numbing complexity. The Medley documentation set describes all the tools in detail, but it would be unreasonable for us to expect a new user to wade through all of it, so this primer is intended as an introduction, to give you a taste of some of the features.

We developed this primer to provide a starting point for new Medley users, to enhance your excitement and challenge you with the potential before you. We're going to make some assumptions about you. For starters, we're going to assume that you're sitting at a workstation that can run Medley. All of the examples in the book figure that you're going to want to try things out. We're also going to assume that you've had some exposure to Lisp, hopefully Common Lisp.

Medley actually consists of two complete Lisp implementations, Common Lisp and InterLisp. All the screen I/O and some of the system functions are in InterLisp. However, thanks to the package system, you can call back and forth between the two languages by simply including a package delimiter in front of a symbol name. This sounds complicated, but it will become clearer once we do some examples.

Throughout we make reference to the *lnterlisp-D Reference Manual* by section and page number. The material in the primer is just an introduction. When you need more depth, use the detailed treatment provided in the manual.

While only you can plot your ultimate destination, you will flnd this primer indispensable for clearly defining and guiding you to the first landmarks on your way.

Acknowledgements

The early inspiration and model for this primer came from the Intelligent Tutoring Systems group and the Learning Research and Development Center at the University of Pittsburgh. We gratefully acknowledge their pioneering contribution to more effective artificial intelligence.

This primer was originally developed by Computer Possibilities, a company committed to making Al technology available. Primary development and writing was done by Cynthia Cosic, with technical writing support provided by Sam Zordich. It has been re-done by Venue staff to reflect changes in the environment since the original publication.

At Xerox Artificial Intelligence Systems, John Vittal managed and directed the project. Substantial assistance was provided by many members of the AlS staff who provided both editorial and systems support.

[This page intentionally left blank]

2. TYPING SHORTCUTS

Once you have logged in to Medley, you are in Lisp. The functions you type into the Executive Window will now execute, that is, perform the designated task. Lisp is case-sensitive; it often matters whether text is typed in upper- or lowercase letters. Use the Shift-Lock key on your keyboard to ensure that everything typed is in capital letters.

You must type all Lisp functions in parentheses. The Lisp interpreter will read from the left parenthesis to the closing right parenthesis to determine both the function you want to execute and the arguments to that function. Executing this function is called "evaluation." When the function is evaluated, it returns a value, which is then printed in the Executive Window. This entire process is called the read-eval-print loop, and is how most Lisp interpreters, including the one for Lisp, run.

The prompt in is a number followed by a left-pointing arrow (see Figure 2.3). This number is the function's position on the History List—a list that stores your interactions with the Lisp interpreter. Type the function `(PLUS 3 4)`, and notice the History List assigns to the function (the number immediately to the left of the arrow). Lisp reads in the function and its arguments, evaluates the function, and then prints the number 7.

Programmer's Assistant

In addition to this read-eval-print loop, there is also a "programmer's assistant." It is the programmer's assistant that prints the number as part of the prompt in the executive window, and uses these numbers to reference the function calls typed after them.

When you issue commands to the programmer's assistant, you will not use parentheses as you do with ordinary functiion calls. You simply type the command, and some specification that indicates which item on the history list the command refers to. Some programmer's assistant commands are `FIX`, `REDO`, and `UNDO`. They are explained in detail below.

Programmer's assistant commands are useful only at the Lisp top level, that is, when you are typing into the Executive Window. They do not work in user-defined functions.

As an example use of the programmer's assistant, use `REDO` to redo your function call `(PLUS 3 4)`. Type **`REDO`** at the prompt (programmer's assistant commands can be typed in either upper- or lowercase) , then specify the previous expression in one of the following ways:

- When you originally typed in the function you now want to refer to, there was a History List number to the left of the arrow in the prompt. Type this number after the programmer's assistant command. This is the method illustrated in Figure 2-1.

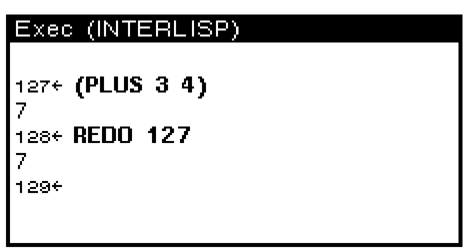


Figure 2-1. Using a Programmer's Assistant Command to `REDO` a Function

- A negative number will specify the function call typed in that number of prompts dago. In this example, you would type in -1, the position immediately before the current position. This is shown in Figure 2-2.

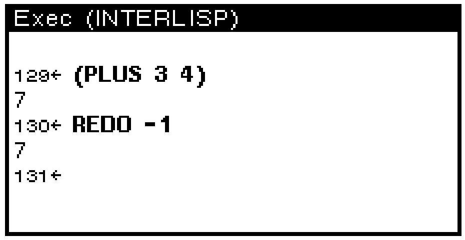


Figure 2-2. Using a Negative Number after the Programmer's Assistant Command

- You can also specify the function for the programmer's assistant with one of the items that was in that function call. The programmer's assistant will search backwards in the History List, and use the first function it finds that includes that item. For example, type **REDO PLUS** to have the functiion `(PLUS 3 4)` reevaluated.

- If you type a programmer's assistant cmmand without specifying a function (i.e., simply typing the command, following by a Return), the programmer's assistant executes the command using the function entered at the previous prompt.

Figure 2-3 shows a few more examples of how to use the programmer's assistant.

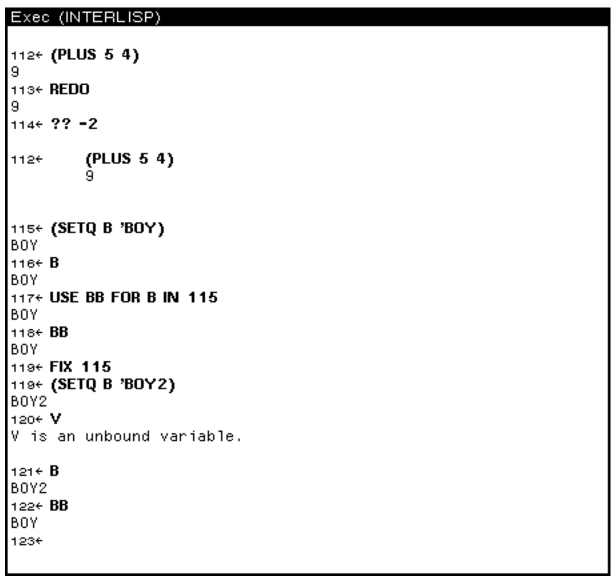


Figure 2-3. Some Applications of the Programmer's Assistant

If You Make a Mistake

Editing in the Executive Window is explained in detail in Chapter 7. In the following section, only a few of the most useful commands are repeated.

To move the caret to a new place in the command being typed, point the mouse cursor at the appropriate position. Then press the left mouse button.

To move the caret back to the end of the command being typed, press Control-X (hold the Control key down, and type **X**).

To delete:

Character behind the caret	Press the Backspace key
Word behind the caret	Press Control-W (hold the Control key down and type W)
Any part of the command	Move the caret to the appropriate place in the command. Hold the right mouse button down and move the the mouse cursor over the text. All of the blackened text between the caret and mouse cursor is deleted when you release the right mouse button.
Entire command	Press Control-U (hold the Control key down and type **U**)

Deletions can be undone. Just press the UNDO key.

To add more text to the line, move the carent to the appropriate position and start to type. Whatever you type will appear at the caret.

[This page intentionally left blank]

TABLE of CONTENTS

15. Displaystreams

16. Fonts

17. The Inspector

18. Masterscope

19. Where Does All the Time Go? SPY

20. Free Menus

[This page intentionally left blank]

3. USING MENUS

The purpose of this chapter is to show you how to use menus. Many things can be done more easily using menus, and there are many different menus provided in the Medley environment. Some are "pop-up" menus that are only available until a selection is made, then disappear until they are needed again. An example of one of these is the Background Menu that appears when the mouse is not in any window and the right mouse button is pressed. A background menu is shown in Figure 3-1. Your background menu may have different items on it.

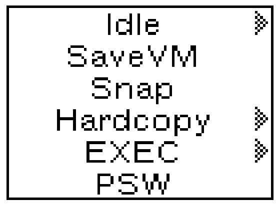


Figure 3-1. Background Menu

Another common pop-up menu is the right button default window menu. This menu is explained more in Chapter 6.

Other menus are more permanent, such as the menu that is always available for use with the Filebrowser. This menu is shown in Figure 3-2., and the specifics of its use with the filebrowser are explained in Chapter 5.

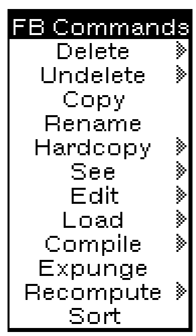


Figure 3-2. Filebrowser Menu

Making a Selection from a Menu

To make a selection from a menu, point with the mouse to the item you would like to select. If one of the mouse buttons is already pressed, the menu item should be highlighted in reverse video. If it is a permanent menu, you must press the left mouse button to highlight the item. When you release the button,m the item will be selected. Figure 3-3 shows a menu with the item "Undo" chosen.

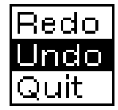


Figure 3-3. Menu with the Item "Undo" Chosen

Explanation of Menu Items

Many menu items have explanations associated with the. If you are not sure what the consequences of choosing a particular menu iem will be, highlight the menu item but do not releast the left mouse button. If the menu item has an explanation associated with it, the explanation will be printed in the prompt window. Figure 3-4 shows the explanation associated with the item "Snap" from the background menu.

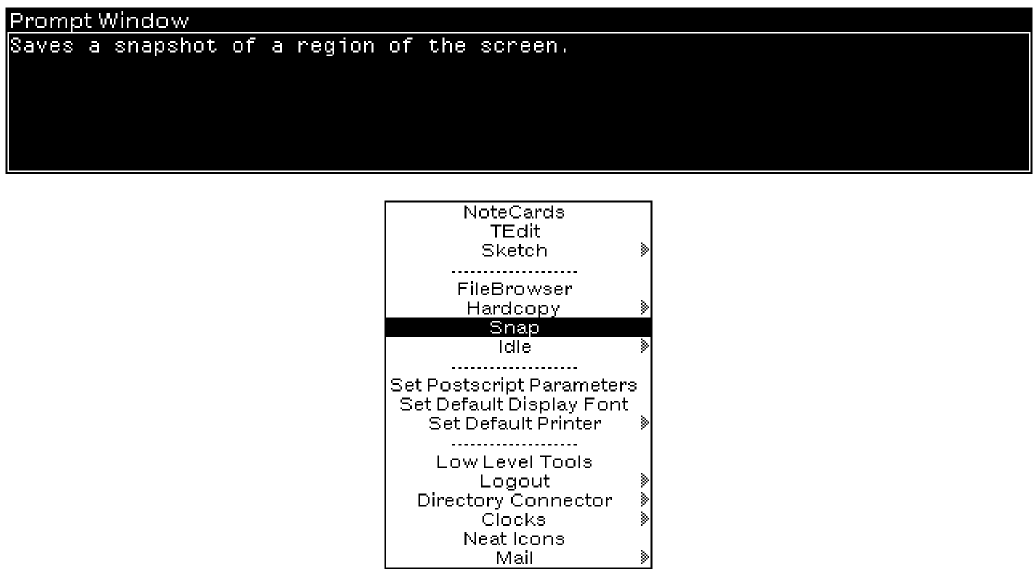


Figure 3-4. Explanation Associated with Selected Menu Item

Submenus

Some menu items have submenus associated with them. This means that, for these items, you can make even more precise choices if you would like to.

A submenu can also be found as described below.

As shown in Figure 3-5, a submenu can be indicated by a gray arrow to the right of the menu item. To see the submenu, highlight the menu item and move the mose to the right to follow the arrow. Choosing an item from a submenu is done the same way you make a choice from the menu. Any submenus that might be associated with the items in the submenu are indicated in the same way as the submenus associated with the items in the menu.

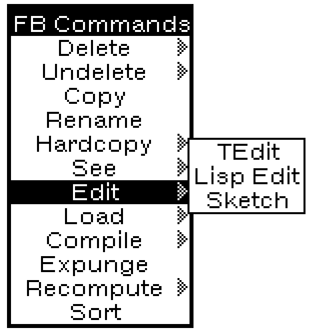


Figure 3-5. Edit Submenu Displayed with Right Arrow

Summary

In summary, here are a few rules of thumb to remember about the interactions of the mouse and system menus:

- Press the left mouse button to select a menu item
- Press the middle mouse button to get more options on a submenu
- Press the right mouse button to see the default right button window menu, and the background menu

[This page intentionally left blank]

4. HOW TO USE FILES

Types of Files

A program file, or Lisp file, contains a series of expressions that can be read and evaluated by the Lisp interpreter. These expressions can include function or macro definitions, variables and their values, properties of variables, and so on. How to save Interlisp-D expressions on these files is explained in Chapter 7. Loading a file is explained in the Simple Commands for Manipulating Files section below.

Not all files, however, have Lisp expressions stored on them. For example, TEdit files store text; sketches are stored on files made with the package Sketch , or can be incorporated into TEdit files. These files are not loaded directly into the environment, but are accessed with the package used to create them, such as TEdit or Sketch.

When you name a file, there are conventions that you should follow. These conventions allow you to tell the type of file by the extension to its name.

If a file contains:	**Then:**
Lisp expressions	It should not have an extension or have the extension `.LISP`. For example, a file called `MYCODE` should contain Lisp expressions.
Compiled Code	It should have the extension `.LCOM or .DFASL`. For example, a file called `MYCODE.DFASL` should contain compiled code.
A Sketch	Its extension should be `.SKETCH`. For example, a file called `MOUNTAINS.SKETCH` should contain a Sketch.
Text	It should have the extension `.TEDIT`. For example, a file called `REPORT.TEDIT` should contain text that can be edited with the editor `TEDIT`.

Directories

This section focuses on how you can find files, and how you can easily manipulate files. To see all the files listed on a device, use the function `DIR`. For example, to see what files are stored in your current directory, type:

```
(DIR *.*)
```

Partial directory listings can be gotten by specifying a file name, rather than just a device name. The wildcard character * can be used to match any number of unknown characters. For example, the command `(DIR T*)` will list the names of all files that begin with the letter `T`. An example using the wildcard is shown in Figure 4-1.

```
Exec (INTERLISP)

126← (DIR {DSK}/USERS/PORTER/TMP/D*)

   {DSK}<users>porter>tmp>
DRAFT.TEDIT;1
DRAFT2.TEDIT;1
127←
```

Figure 4-1. Using DIR with a Wildcard

Directory Options

Various words can appear as extra arguments to the DIR command. these words give you extra information about the files.

SIZE displays the size of each file in the directory. For example, type:

```
(DIR {DSK} SIZE)
```

DATE displays the creation date of each file in the directory. An example of this is shown in Figure 4-2.

```
Exec (INTERLISP)

127← (DIR {DSK}/USERS/PORTER/TMP/D* DATE)

                      CREATIONDATE

   {DSK}<users>porter>tmp>
DRAFT.TEDIT;1        28-Jan-92 11:26:21
DRAFT2.TEDIT;1       28-Jan-92 11:26:22
128←
```

Figure 4-2. Example Using DATE

DEL deletes all the files found by the directory command.

Subdirectories

Sudirectories are very helpful for organizing files. A set of files that have a single purpose (for example, all the external documentation files for a system) can be grouped together into a subdirectory.

To associate a subdirectory with a filename, simply include the desired subdirectory as part of the name of the file. Subdirectories are specified after the device name and before the simple filename. The first subdirectory should be between less-than and greater-than signs (angle brackets) < >, with nested subdirectory names only followed by a greater than sign >. For example:

```
{DSK}<Directory>SubDirectory>SubSubDirectory>...>filename
```

or use the UNIX convention:

```
{DSK}/Directory/Subdirectory/Subsubdirectory/filename
```

To See What Files Are Loaded

If you type FILELST<CR>, the names of all the files you loaded will be displayed.

Type SYSFILES<CR> to see what files are loaded to create the sysout.

Simple Commands for Manipulating Files

When using these functions, always be sure to specify the full filename, including subfile directories if appropriate.

To have the conents of a file displayed in a window:

(SEE '*filename*)

To copy a file (see Figure 4-3):

(COPYFILE '*oldfilename* '*newfilename*)

Figure 4-3. Example Use of COPYFILE

To delete a file (see Figure 4-4):

(DELFILE '*filename*)

Figure 4-4. Example Use of DELFILE

To rename a file:

(RENAMEFILE '*oldfilename* '*newfilename*)

Files that contain Lisp expressions can be loaded into the environment. That means that the information on them is read, evaluated, and incorporated into the Medley environment. To load a file, type:

(LOAD '*filename*)

Connecting to a Directory

Often, each person or project has a subdirectory where files are stored. If this is your situation, you will want any files you create to be put into this directory automatically. This means you should "connect" to the directory.

CONN is the Medley command that connects you to a directory. For example, CONN in Figure 4-5 connects you to the subsubdirectory IM, in the subdirectory PRIMER, the directory LISPFILES, on the device DSK. This information—the device and the directory names down to the subdirectory to which you want to be connected—is called the "path" to that subdirectory. CONN expects the path to a directory as an argument.

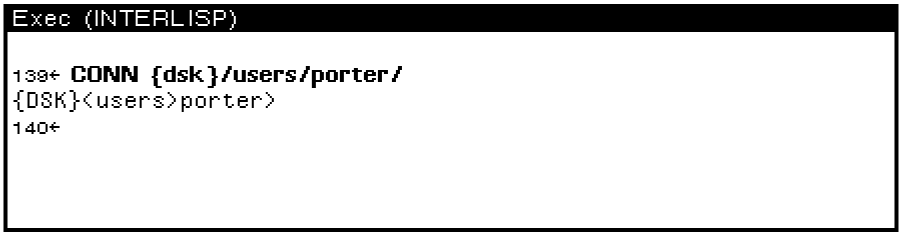


Figure 4-5. CONNecting to Subdirectory Primer Subsubdirectory IM

Once you are connected to a directory, the command DIR will assume you want to see the files in that directory, or any of its subdirectories.

Other commands that require a filename as an argument (e.g., SEE, above) will assume that the file is in the connecteds directory if there is no path specified with the filename. This will often save you typing.

File Version Numbers

When stored, each filename is fillowed by a semicolon and a number, as shown in this example:

```
MYFILE.TEDIT;1
```

The number is the version number of the file. This is the system's way of protecting your files from being overwritten. Each time the file is written, a new file is created with a version number one greater than the last. This new file will have everything from your previous file, plus all of your changes.

In most cases, you can exclude the version number when referencing the file. When the version is not specified, and there is more than one version of the file on that particular directory, the system generally uses your most recent version. An exception is the function DELFILE, which deletes the oldest version (the one with the lowest version number) if none is specified.

[This page intentionally left blank]

5. FILEBROWSER

The FileBrowser is a Lisp Library Package that works with files stored on disk and floppy devices, and can be used as a file directory editor. If it is not loaded into your sysout, you need to load it first by typing:

```
(LOAD 'FILEBROWSER.LCOM)
```

Calling the FileBrowser

Calling the FileBrowser with the directory calls up the files stored in that directory:

```
(FB '<usr>local>lde>)
```

Another way to call a FileBrowser is to choose "FileBrowser" from the background menu. You will be prompted for a description of the files to be included (see Figure 5-1). Type an asterisk (*), then press Return to see all the files in the connected directory.

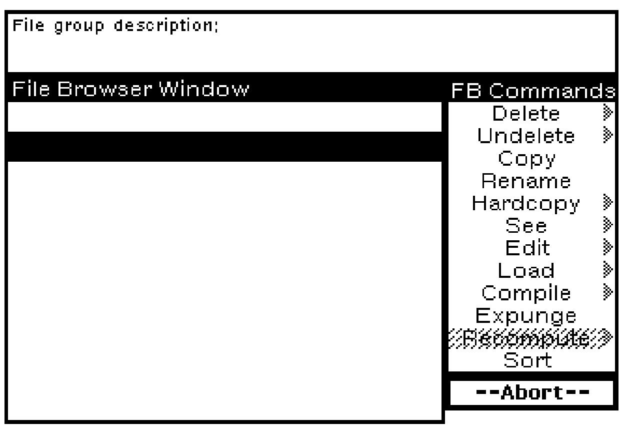


Figure 5-1. Prompt for Files to Include in FileBrowser

These show a directory of the device in a window you can leave on the screen at all times. The parts of the FileBrowser window are shown below.

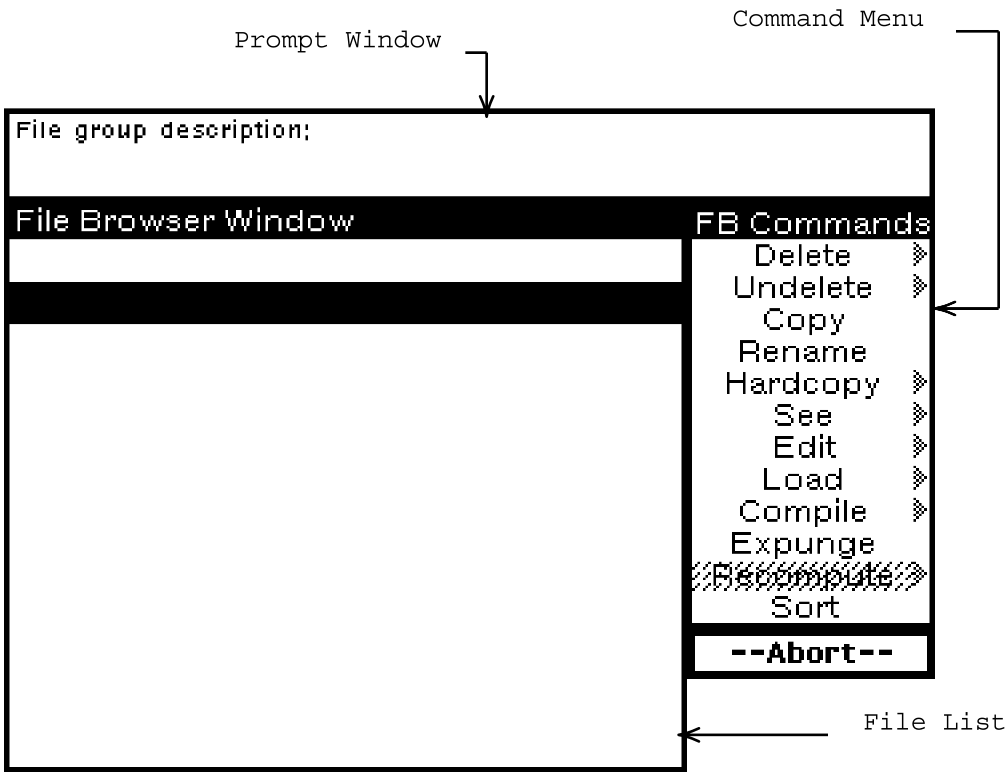


Figure 5-2. Parts of a FileBrowser Window

Now you do not need to continually type the directory command.

To use the FileBrowser, choose a file by pointing to the file with the mouse and pressing the left or middle mouse button. A small dark arrow appears to the left of the file name. Choose a command from the menu at the right. In Figure 5-3, the files `OCH1.TEDIT;1`, `OCH10.TEDIT;1`, and `OCH11.TEDIT;1` have been selected.

The left mouse button only allows you to choose one file at a time. Even if you choose other files, only the last file you selected with the left mouse button will remain marked as chosen. When you use the middle mouse button to select a file, the file is added to those already chosen.

To unpick an already chosen file, hold the Control key down while pressing the middle mouse button.

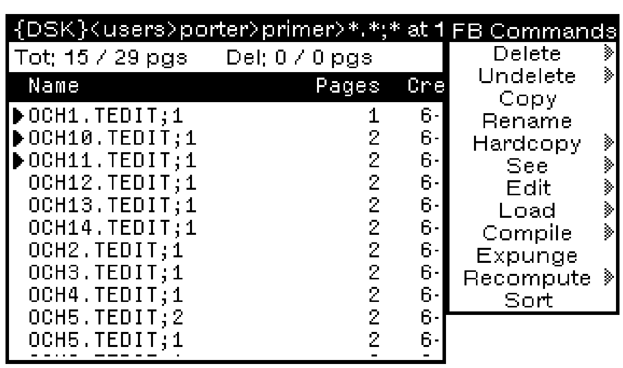


Figure 5-3. Files Chosen

The next section contains a summary of the FileBrowser commands.

FileBrowser Commands

`Delete` In the FileBrowser, this command marks a file, or files, for deletion (see Figure 5-4). These files are marked by a black line crossing through them. You may select and mark any number of files for deletion. `Delete` does not actually remove these files from the device. The `Expunge` command actually wipes out the files previously marked for deletion.

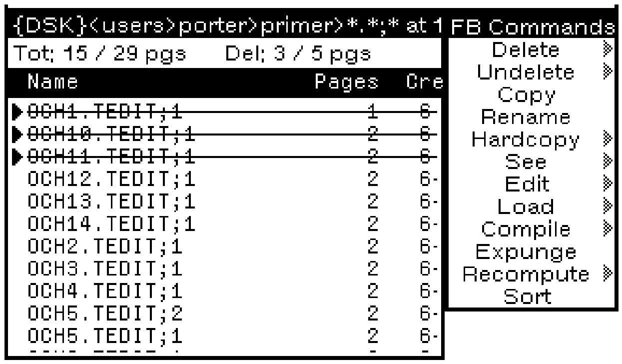


Figure 5-4. Files Marked for Deletion

`Undelete` Undoes the delete command for one or more files. Undelete erases the black line through a file marked for deletion.

`Copy` This command copies the chosen file. The destination filename should be typed at a prompt that appears in the window above the FileBrowser. Wildcards do not work for this prompt. You must type the whole unquoted filename. If more than one file is chosen to be copied, you will be prompted for a directory name. The files will be copied into the directory you give, but with the same filenames as the ones they have in their original location.

`Rename` This command works much like the Copy command, but does not leave the original file. The chosen file will be renamed to the destination filename. You will be prompted, in the prompt window, for the destination filename. Give the complete unquoted filename. If more than one file is chose to be renamed, you will be prompted for a directory name. The files will be moved into the directory you give.

`Hardcopy` If you do not have a hardcopy device, using this command causes an error. Otherwise, it gives a hardcopy of the file.

`See` Shows you a file in a window. To use this command, choose a single filename, then the See command. You are prompted for a window. Each time the See command is chosen, a new window is opened to display the file.

`Edit` Calls the editor with the file as input. If the file is an executable one (i.e., Lisp code as opposed to a documentation file), only the `FILECOMS` list is edited. The `FILECOMS` list is the list of variables, lists, and functions that are contained on that file. FileBrowser loads it and then allows you to edit the `FILECOMS`.

`Load` Choose a file with the left mouse button, or a group of files with the middle mouse button. Once the filenames have been blackened, choose the Load command to load them all into Medley.

`Compile` This command calls the file compiler with the chosen filename(s) as arguments. The compiler compiles a file found on a storage device ({`DSK`}), not the functions defined in the Medley image. If any functions on a loaded file have been changed, run the function (`MAKEFILE 'filename`) to write the current version before compiling it. Files do not have to be loaded to use the `Compile` command.

`Expunge` This command completely deletes all the marked files from the directory. This allows you to remove unwanted files from your storage device.

`Recompute` Choose this command when you know that the directory has been changed and should be reread (e.g., after creating new versions of a file).

6. THOSE WONDERFUL WINDOWS!

A window is a designated area on the screen. Every rectangular box on the screen is a window. While Medley supplies many of the windows (such as the Executive Window), you may also create your own. Among other things, you will type, draw pictures, and save portions of your screen with windows.

Windows Provided by Medley

Two important windows are available as soon as you enter the Medley environment. One is the Executive Window, the main window where you will run your functions. It is the window that the caret is in when you turn on your machine, and load Medley. It is shown in Figure 6-1.

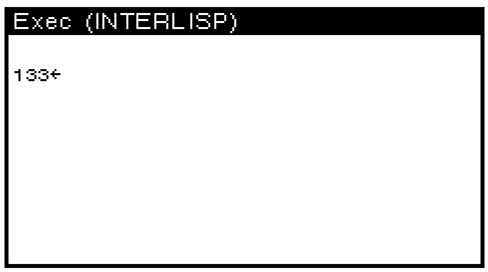


Figure 6-1. Medley Executive Window

The other window that is open when you enter Medley is the "Prompt Window". It is the long thin black window at the top of the screen. It displays system prompts, or prompts you have associated with your programs. (See Figure 6-2.)

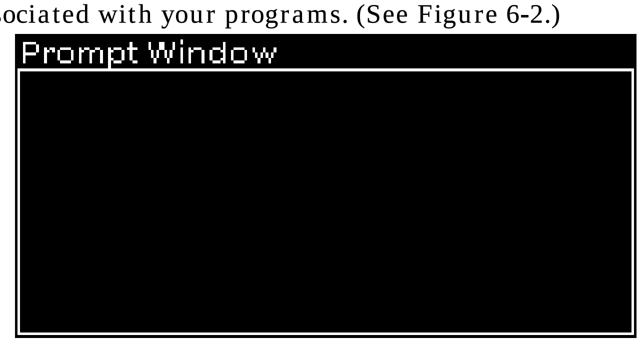


Figure 6-2. Prompt Window

Other programs, such as the editors, also use windows. These windows appear when the program starts to run, and close (no longer appear on the screen) when the program is done running.

Creating a Window

To create a new window, type: **(CREATEW)**. The mouse cursor will change, and have a small square attached to it. (See Figure 6-3.)

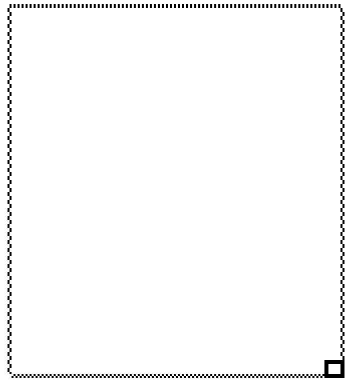


Figure 6-3. Mouse Cursor Asking You to Sweep Out Window

There may be a prompt in the prompt window to create a window. Press and hold the left mouse button. Move the mouse around, and notice that it sweeps out a rectangle. When the rectangle is the size that you'd like your window to be, release the left mouse button. More specific information about the creation of windows, such as giving them titles and specifying their size and position on the screen when they are created, is given in the WINDOWPROP section of Chapter 12.

Right Button Default Window Menu

Position the cursor inside the window you just created, and press and hold the right mouse button. A menu of commands should appear (do not release the right button!), like the one in Figure 6-4. To execute one of the commands on this menu, choose the item. Making a choice from a menu is explained in Chapter 3.

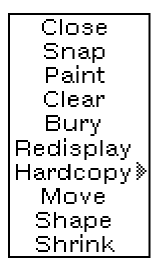


Figure 6-4 Right Button Default Window Menu

As an example, select "Move" from this menu. The mouse cursor will become a ghost window (just an outline of a window, the same size as the one you are moving), with a square attached to one corner, like the one shown in Figure 6-5.

Figure 6-5 Mouse Cursor for Moving a Window

Move the mouse around. The ghost window will follow. Click the left mouse button to place tho window in a new location.

Choose "Shape", and notice that you are prompted to sweep out another window. Your original window will have the shape of the window you sketch out.

Explanation of Each Menu Item

The meaning of each right button default window menu item is explained below:

Close	Removes the window from the screen
Snap	Copies a portion of the screen into a new window
Paint	Allows drawing in a window
Clear	Clears the window by erasing everything within the window boundaries
Bury	Puts the window beneath all other windows that overlap it
Redisplay	Redisplays the window contents
Hardcopy	Sends the contents of the window to a printer or to a flle
Move	Allows the wi ndow to be moved to a new spot on the screen
Shape	Repositions and/or reshapes the window
Shrink	Reduces the window to a small black rectangle called an icon, or, if appropriate, to the shape for that window type (see Figure 6-6).

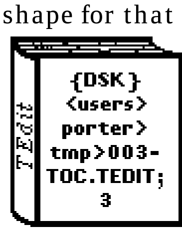


Figure 6-6 Example Icon

Expand	Changes an icon back to its original window. Position the mouse cursor on the icon, depress the right button, and select Expand. Or, just button the icon with the middle mouse button.

These right-button default window menu selections are available in most windows, including the Executive Window. When the right button has other functions in a window (as in an editor window), the right button default window menu should be accessible by pressing the Right button in the black border at the top of the window.

Scrollable Windows

Some windows in Medley are "scrollable". This means that you can move the contents of the window up and down, or side to side, to see anything that doesn't fit in the window.

Point the mouse cursor to the left or bottom border of a window. If the window is scrollable, a "scroll bar" will appear. The mouse cursor will change to a double headed arrow. (See Figure 6-7.)

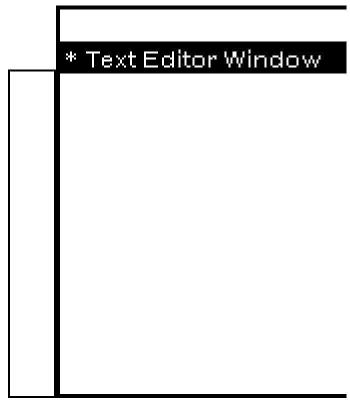


Figure 6-7. Scroll Bar of Scrollable Window

The scroll bar represents the full contents of the window. The example scroll bar is completely white because the window has nothing in it When a part of the scroll bar is shaded, the amount shaded represents the amount of the window's contents currently shown. If everything is showing, the scroll bar will be fully shaded. (See Figure 6-8.) The position of the shading is also important. It represents the relationship of the section currently diplayed to the the full contents of the window. For example, if the shaded section is at the bottom of the scroll bar, you are looking at the end of the file.

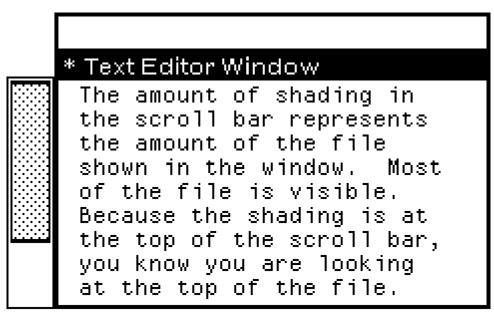


Figure 6-8 Top of File When Shading at Top of Scroll Bar

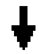
When the scroll bar is visible, you can control the section of the window's contents displayed:

- To move the contents higher in the window (scroll the contents up in the window), press the leff button of the mouse, the mouse cursor changes to look like this:

Figure 6-9. Upward Scrolling Cursor

The contents of the window will scroll up, making the line thit the cursor is beside the topmost line in the window.

- To move the contonts lower in the window (scroll the contents "down" in the window), press the right button of the mouse, and the mouse cursor changes to look like this:



Flgure 6-10. Downward Scrolling Cursor

The contents of the window scroll down, moving the line that is the topmost line in the window to beside the curtor.

- To show a specific section of the window’s contents, remember that the scroll bar represents the full contents of the window. Move the mouse cursor to the relative position of the section you want to see (e.g., to the top of the scroll bar if you want to see the top of the window’s contents). Press the middle button of the mouse. The mouse cursor will look like this:

Figure 6-11 Proportional Scrolling Cursor

When you release the middle mouse button, the window’s contents at that relative position will be displayed.

The position of the mouse in the scroll bar defines how much of the window will be scrolled. If it is near the top, then only a little will be scrolled. If it is near the bottom, most of the window will be scrolled.

Other Window Functions

PROMPTPRINT

Prints an expression to the black prompt window.

For example, type

(PROMPTPRINT "THIS WILL BE PRINTED IN THE PROMPT WINDOW")

The message will appear in the prompt window. (See Figure 6-12.)

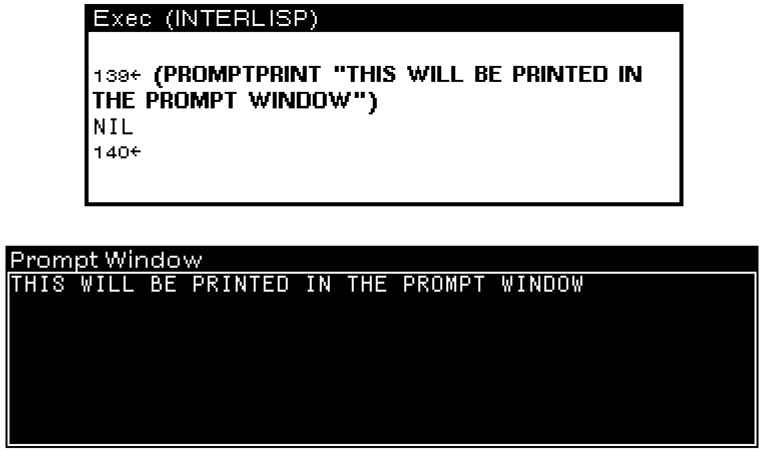


Figure 6-12 PROMPTPRINTing

WHICHW

Returns as a value the name of the window that the mouse cursor IS in.

`(WHICHW)` can be used as an argument to any function expecting a window, or to reclaim a window that has no name (that is not attached to some particular part of the program.).

7. EDITING AND SAVING

This chapter explains how to define functions, how to edit them, and how to save your work.

Defining Functions

DEFUN can be used to define new functions. The syntax for it is:

(DEFUN (<functionname> (<parameter-list><body-of-function>))

New functions can be created with DEFUN by typing directly into the Executive Window. Once defined, a function is a part of the Medley environment. For example, the function EXAMPLE-ADDER is defined in Figure 7-1.

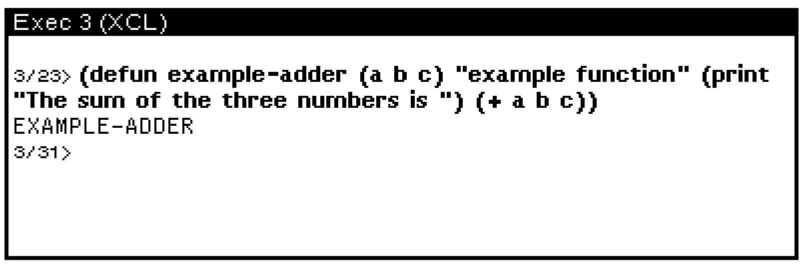


Figure 7-1. Defining the Function EXAMPLE-ADDER

Now that the function is defined, it can be called from the Executive Window:

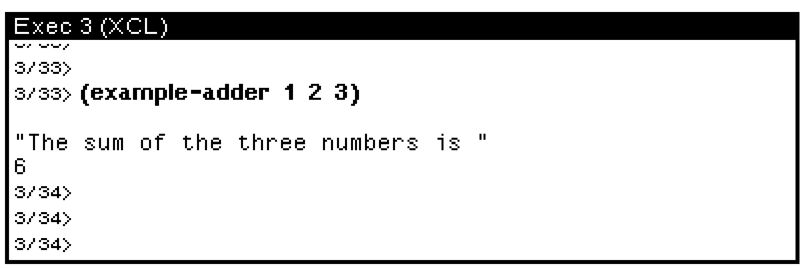


Figure 7-2.. After EXAMPLE-ADDER is defined, it can he executed

The function returns 6, after printing out the message.

Functions can also be defined using the editor DEdit described above. To do this, simply type

(ED *function-name* 'FUNCTIONS)

You will be told that no definition exists for the function, and a menu will pop up asking you what type of function you would like to create:

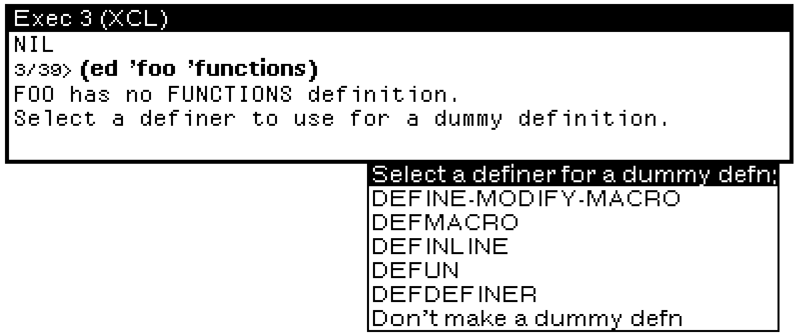


Figure 7-3 Selecting a Function Template

Selecting the appropriate type will pop up an editor window with a function template. The use of the editor is explained in the Using the List Structure Editor section below.

Simple Editing in the Executive Window

First, type in an example function to edit:

```
3/41>  (defun your-first-function (a b)
          (if (> a b)
               '(the first is greater)
               '(the second is greater)))
```

To run the function, type:

```
3/42>  (YOUR-FIRST-FUNCTION 3 5)
(THE SECOND IS GREATER)
```

Now, let's alter this. Type:

```
3/43> FIX 41
```

Note that your original function is redisplayed, and ready to edit. (SeeFigure 7-4.)

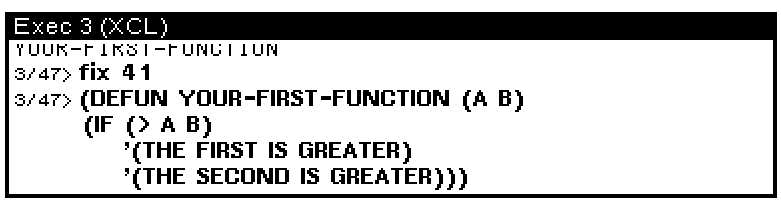


Figure 7-4. Using FIX to Edit a Fundion

Move the text cursor to the appropriate place in the function by positioning the mouse cursor and pressing the left mouse button.

Delete text by moving the caret to the beginning of the section to be deleted. Hold the right mouse button down and move the mouse cursor over the text. All of the highlighted text between the caret and mouse cursor is deleted when you release the right mouse button.

If you make a mistake, deletions can be undone. Press the UNDO key on the keypad to the left of the keyboard.

Now change `GREATER` to `BIGGER`:

1. Position the mouse cursor on the G of GREATER, and click the left mouse button. The text cursor is now where the mouse cursor is.
2. Next, press the right mouse button and hold it down. Notice that if you move the mouse cursor around, it will blacken the characters from the text cursor to the mouse cursor. Move the mouse so that the word "GREATER" is highlighted.
3. Release the right mouse button and GREATER is deleted.
4. Without moving the cursor, type in BIGGER.
5. There are two ways to end the editing session and run the function. One is to type Control-**X**. (Hold the Control key down, and type **X**.) Another is to move the text cursor to the end of the line and crø In both cases, the function has been edited!

Try the new version of the function by typing:

```
3/48> (YOUR-FIRST-FUNCTION 8 9)
(THE SECOND IS BIGGER)
```

and get the new result, or you can type:

```
3/49> REDO 42
(THE SECOND IS BIGGER)
```

Using the List Structure Editor

If the function you want to edit is not readily available (i.e. the function is not in the Executive Window, and you can't remember the history list number, or you simply have a lot of editing), use the List Structure Editor, often called SEdit. This editor is evoked with a call to ED:

```
81←(ED 'YOUR-FIRST-FUNCTION 'FUNCTIONS)
```

Your function will be displayed in an edit window, as in Figure 7-5.

If there is no edit window on the screen, you will be prompted to create a window. As before, hold the leff mouse button down, move the mouse until it forms a rectangle of an acceptable size and shape, then release the button. Your function definition will automatically appear in this edit window.

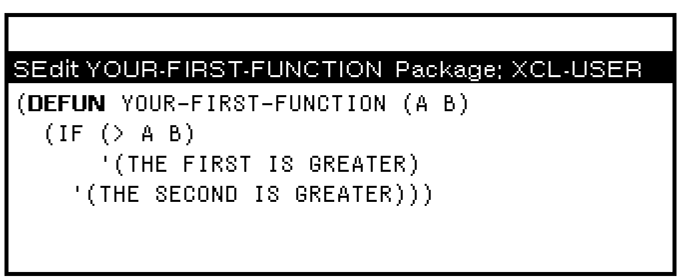
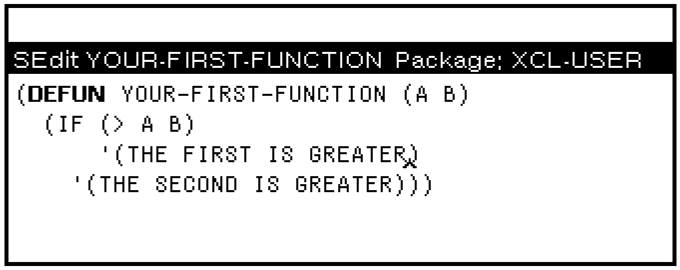


Figure 7-5. An Edit Window

Many changes are easily done with the structure editor. Notice that by pressing the left mouse button you can place the caret in position, and by pressing the middle mouse button you can select atoms or s-expressions. Repeated pressing of the middle button selects bigger pieces of text.

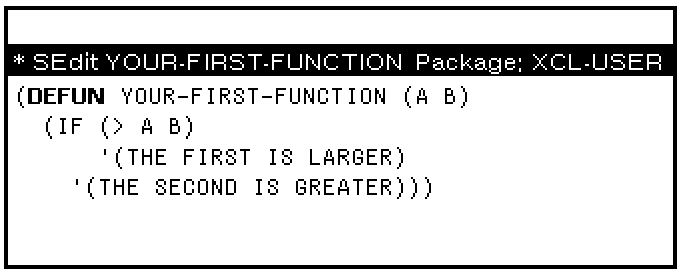
To add an expression that does not appear in the edit window (i.e., it cannot simply be underlined), place the caret at the insertion point and type it in.. For example, to replace the first GREATER with LARGER, place the caret to the left of GREATER, as shown in Figure 7-6.



```
SEdit YOUR-FIRST-FUNCTION  Package: XCL-USER
(DEFUN YOUR-FIRST-FUNCTION (A B)
  (IF (> A B)
      '(THE FIRST IS GREATER)
    '(THE SECOND IS GREATER)))
```

Figure 7-6. Caret Placement Prior to Changing GREATER with LARGER

Now press the DELETE key seven times, and type in LARGER. The window now looks like this:



```
* SEdit YOUR-FIRST-FUNCTION  Package: XCL-USER
(DEFUN YOUR-FIRST-FUNCTION (A B)
  (IF (> A B)
      '(THE FIRST IS LARGER)
    '(THE SECOND IS GREATER)))
```

Figure 7-7. GREATER Changed to LARGER

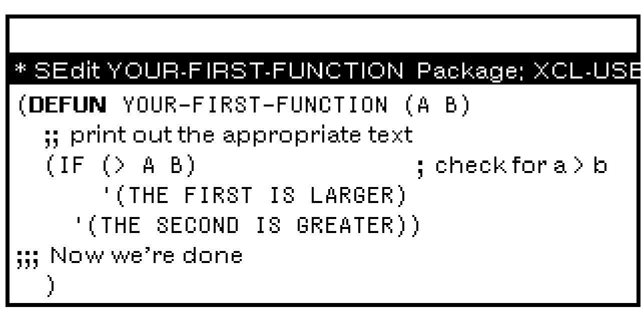
Notice the asterisk in the left edge of the title bar of the window. This designates that the function has be changed. Now exit the edit session by typing Control-X, and the function will be redifined.

Commenting Functions

Text can be marked as a comment by typing a semi-colon before the text of the comment.

```
; This is the form of a comment
```

Inside an editor window, the comment will be printed in a different font and may be moved to the far right of the code. SEdit is familiar with the Common Lisp convention of single comments being on the far right, double comments being justified with the function level, and triple comments being on the far left, as is shown in Figure 7-8.



```
* SEdit YOUR-FIRST-FUNCTION  Package: XCL-USE
(DEFUN YOUR-FIRST-FUNCTION (A B)
  ;; print out the appropriate text
  (IF (> A B)                 ; check for a > b
      '(THE FIRST IS LARGER)
    '(THE SECOND IS GREATER))
;;; Now we're done
  )
```

Figure 7-8. Placement of Comments

There are other editor commands which can be very useful. To learn about them, read Appendix B of the *Release Notes*.

File Functions and Variables: How to See and Save Them

With Medley, all work is done inside the Lisp environment. There is no operating system or command level other than the Executive Window. All functions and data structures are defined and edited using normal Lisp commands. This sertion describes tools in the Medley environment that will keep track of any changes that you make in the environment that you have not yet saved on files, such as defining new functions, changing the values of variables, or adding new variables. And it then has you save the changes in a file you specify. All of these functions are in the `INTERLISP` (`IL:`) package.

File Variables

Certain system-defined global variables are used by the file package to keep track of the environment as it stands. You can get system information by checking the values of these variables. Two important variables follow.

- `FILELST` evaluates to a list, all files that yoU have loaded into the Medley environment.
- *filename*`COMS` (Each file loaded into the Lisp environment has associated with it a global variable, whose name is formed by appending `COMS` to the end of the filename.) This variable evaluates to a list of all the functions, variables, bitmaps, windows, and soon, that are stored on that particular file.

  For example, if you type:

  ```
  MYFILECOMS
  ```

  the system will respond with something like:

  ```
  ((FNS YOUR-FIRST-FUNCTION  )
   VARS))
  ```

Saving Interlisp-D on Files

The functions `(FILES?)` and `(MAKEFILE '`*filename*`)` are useful when it is time to save function, variables, windows, bitmaps, records and whatever else to files.

`(FILES?)` displays a list of variables that have values and are not already a part of any file, and then the functions that are not already part of any file.

Type:

`(FILES?)`

the system will respond with something like:

```
the variables: MY.VARIABLE CURRENT.TURTLE...to be
dumped
```

```
the functions: RIGHT LEFT FORWARD BACKWARD
CLEAR-SCREEN...to be dumped

want to say where the above go?
```

If you type Y, the system will prompt with each item. There are three options:

1. To save the item, type the filename (unquoted) of the file where the item should be placed. (This can be a brand new file or an existing file.)
2. To skip the item, without removing it from consideration the next time (FILES?) is called, type crø This will allow you to postpone the decision about where to save the item.
3. If the item should not be saved at all, type]. Nowhere will appear after the item.

Part of an example interaction is shown in the following figure:

```
Exec (INTERLISP)

57← (FILES?)
To be dumped:
NEWFILE ...changes to VARS: NEWFILECOMS
                      FNS: TEST
    plus the Common Lisp structures: MyStruct
    plus the functions: Function
want to say where the above go ? yes
(Common Lisp structures)
MyStruct  Nowhere
(functions)
Function  File name: NFILE
create new file NFILE ? yes
To be compiled: FOREIGN-FUNCTIONS, FOO
To be listed: PAINTW, FOREIGN-FUNCTIONS, FOO, COURIERSERVE
58←
```

Figure 7-9. Part of an interaction using the function FILES?

(FILES?) assembles the items by adding them to the appropriate file's COMS variable (see the File Variables section above). (FILES?) does NOT write the file to secondary storage (disks or floppies). It only upclates the global variables discussed in the File Variables section above.

(MAKEFILE 'filename)

actually writes the file to secondary storage.

Type:

(MAKEFILE 'MY.FILE.NAME)

and the system will create the file. The function returns the full name of the file created. (i.e. {DSK}MY.FlLE.NAME.; 1).

Files written to {DSK} are permanent files. They can be removed only by the user deleting them or by reformatting the disk.

Other file manipulation functions can be found in Chapter 4.

8. YOUR INIT FILE

Lisp has a number of global variables that control the environment. Global variables make it easy to customize the environment to fit your needs. One way to do this is to develop an INIT file. This is a file that is loaded when you start an image. You can use it to set variables, load files, define functions, and any other things that you want to do to make the Medley environment suit you.

Using the USERGREETFILES Variable

As described in File Variables section of Chapter 11, each program file has a global Your INIT file could be called INIT, INIT.LISP, INIT.USER, or whatever the convention is at your site. There is no default name preferred by the system, it just looks for the files listed in the variable USERGREETFILES (see below). Check to see what the preference is at your site. Put this file in your directory. Your directory name should be the same as your login name. The INIT file is loaded by the function GREET. GREET is normally run when Medley is started. If this is not the case at your site, or you want to use the machine and Medley has already been started, you can run the function GREET yourself. If your user name was, for example, TURING, then you would type:

```
(GREET 'TURING)
```

This does a number of things, including undoing any previous greeting operation, loading the site init file, and loading your init file. Where GREET looks for your INIT file depends on the value of the variable USERGREETFILES. The value of this variable is set when the system's SYSOUT file is made, so check its value at your site! For example, its value could be:

```
60← USERGREETFILES
(({DSK}INIT %. COM)
 ({DSK}INIT- USER %. COM)
 ({DSK}INIT- USER)
 ({DSK}INIT))
```

Figure 8-1. Possible Value of USERGREETFILES

In each place you see >USER>, the argument passed to GREET is substituted into the path. This is your login name if you are just starting Medley. For example, the first value in the list would have the system check to see whether there was a {DSK}<LISPFlLES>TURING>INIT.LISP file. No error is generated if you do not have an INIT file, and none of the files in USERGREETFILES are found.

Making an Init File

As described in File Variables section of Chapter 11, each program file has a global variable associated with it, whose name is formed by appending COMS to the end of the root filename. For any of the standard INIT file names, the variable INITCOMS is used. To set up an init file, begin by editing this variable. Type:

```
(DV INITCOMS)
```

An SEdit window wiil appear. This window is the same as the one called with the function DF, and described in the Using the List Structure Editor section in Chapter 7. This chapter assumes that you know how to use the SEdit structure editor .

The COMS variable is a list of lists. The first atom in each internal list specifies for the file package what types of items are in the list, and what it is to do with them. This section will deal with three types of lists: VARS, FILES, and P. Please read about others in Chapter 17 of the *IRM*.

Notice that inside the vars list, there is yet another list. The first item in the list is the name of the variable. It is bound to the value of the second item. There are many other variables that you can set by adding them to the VARS list. Some of these variables are described in Chapter 24, and many others can be found in the *IRM*.

If you want to automatically load files, that can be done in your init file also. For example, if you always want to load tho Library file SPY.LCOM, you can load it by editing tho INITCOMS variable to list the appropriate file in the list starting with FILES:

```
.
.
.
(FILES SPY)
.
.
.
```

Figure 8-2. INITCOMS Changed to Load SPY.LCOM File

Other files can also be added by simply adding their names to this FILES list.

Another list that can appear in a COMS list begins with P. This list contains Lisp expressions that are evaluated when the file is loaded. Do not put DEFINEQ expressions in this list. Define the function in the environment, and then save it on the file in the usual way (see Chapter 7).

One type of expression you might want to see here, however, is a FONTCREATE function (see Chapter 16). For example, of you want to use a Helvetica 12 BOLD font, and there is not a fontdescriptor for it normally in your environment, the appropriate call to FONTCREATE should be in the "P" list. The INITCOMS would look like this:

```
.
.
.
(FILES SPY)
(P (FONTCREATE 'HELVETICA 12 'BOLD))
.
.
.
```

Figure 8-3. INITCOMS Edited to Include a call to FONTCREATE

To quit, exit from SEdit in the usual way. When you run the function MAKEFILES (see Chapter 7), be sure that you are connected to the directory (see Chapter 4) where the INIT file should appear. Now when GREET is run, your Init file will be loaded.

9. MEDLEY FORGIVENESS: DWIM

DWIM (Do What I Mean) is an Interlisp utility that makes life easier.

DWIM tries to match unrecognized variable and function names to known ones. This allows Lisp to interpret minor typing errors or misspellings in a function, without causing a break. Line 152 of Figure 9-1 illustrates how the misspelled BANNANNA was replaced by BANANA before the expression was evaluated.

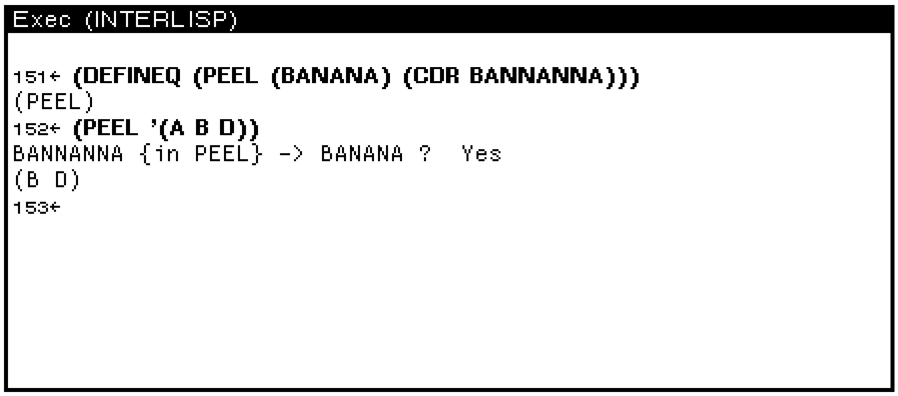


Figure 9-1. Examples of DWIM Features

Sometimes DWIM may alter an expression you didn't want it to. This may occur if, for example, a hyphenated function name (e.g., (MY-FUNCTION)) is misused. If the system does not recognize the function name, it may think you are trying to subtract "FUNCTION" from "MY". DWIM also takes the liberty of updating the function, so it will have to be fixed. However, this is as much a blessing as a curse, since it points out the misused expression!

10. BREAKPACKAGE

The Break Package is a part of Interlisp that makes debugging your programs much easier.

Break Windows

A break is a function either called by the programmer or by the system when an error has occurred. A separate window opens for each break. This window works much like the Executive Window, except for extra menus unique to a break window. Inside a break window, you can examine variables, look at the call stack at the time of the break, or call the editor. Each successive break opens a new window, where you can execute functions without disturbing the original system stack. These windows disappear when you resolve the break and return to a higher level.

Break Package Example

This example illustrates the basic break package functions. A more complete explanation of the breaking functions, and the break package will follow.

The correct definition of FACTORIAL is:

```
(defun factorial (x)
     (if (zerop x)
          1
         (* x (factorial (1- x)))))
```

To demonstrate the break package, we have edited in an error: DUMMY in the IF statement is an unbound atom, it lacks a value.

```
((defun factorial (x)
     (if (zerop x)
          dummy
         (* x (factorial (1- x)))))
```

The evaluated function

```
(FACTORIAL 4)
```

should return 24, but the above function has an error. DUMMY is an unbound atom, an atom without an assigned value, so Lisp will "break". A break window appears (Figure 10-1), that has all the functionality of the typing lisp expressions into the Executive Window (The top level), in addition to the break menu functions. Each consecutive break will move to another level "down".

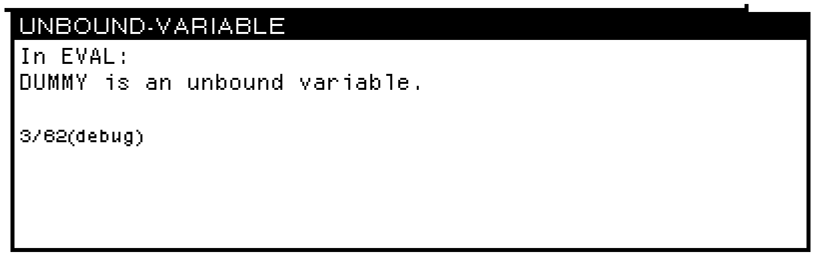


Figure 10-1. Break Window

Move the mouse cursor into the break window and hold down the middle mouse button. The Break Menu will appear. Choose BT. Another menu, called the stack menu, will appear beside the break window. Choosing stack items from this menu will display another window. This window displays the function's local variable bindings, or values (see Figure 10-2). This new window, titled FACTORlAL Frame, is an inspector window (see inspector Chapter 17).

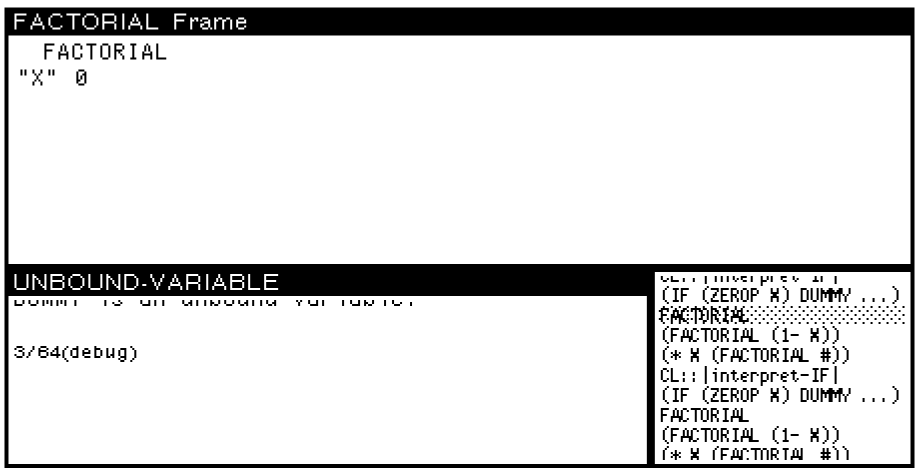


Figure 10-2. Back Trace of the System Stack

From the break window, you can call the editor for the function `FACTORIAL` by `middle-buttoning on the word FACTORIAL and selecting DisplayEdit from the menu that pops up`.

Replace the unbound atom `DUMMY` with 1. Exit the editor .

The function is fixed, and you can restart it from the last call on the stack. (It does not have to be started again from the Top Level.) To begin again from the last call on the stack, choose the last (top) `FACTORIAL` call in the BT menu. Select R`EVERT` from the middle button break window, or type it into the window. The break window will close, and a new one will appear with the message: **`Breakpoint at FACTORIAL`**

To start execution with this last call to `FACTORIAL`, choose OK from the middle button break menu. The break window will disappear, and the correct answer, 24, will be returned to the top level.

Ways to Stop Execution from the Keyboard (Breaking Lisp)

There are ways you can stop execution from the keyboard. They differ in terms of how much of the current operating state is saved:

Control-G Provides you with a menu of processes to interrupt. Your process will usually be "`EXEC`". Choose it to break your process. A break window will then appear.

Control-B Causes your function to break, saves the stack, then displays a break window with all the usual break functions. For information on other interrupt characcers, see Chapter 30 in the *IRM*.

Break Menu

Move the mouse cursor into the break window. Hold the middle button down, and a new menu will pop up, like the one in Figure 10-3.

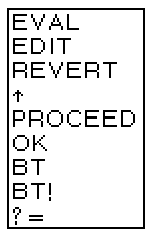


Figure 10-3. Middle Button Menu in Break window

Five of the selections are particularly important when just starting to use Medley:

`BT` Back Trace displays the stack in a menu beside the break window. Back Trace is a very powerful debugging tool. Each function call is placed on the stack and removed when the execution of that function is complete. Choosing an item on the stack will open another window displaying that item's local variables and their bindings. This is an inspector window that offers all the power of the inspector. (For details, see the section on the Inspector, Chapter 17.)

`? =` Before you use this menu option, display the stack by choosing BT from this menu, and choose a function from it. Now, choose ?=. It will display the current values of the arguments to the function that has been chosen from the stack.

↑ Move back to the previous break window, or if there is no other break window, back to the top level, the Executive Window.

`REVERT` Move the point of execution back to a specified function call before the error. The function to revert back to is, by default, the last function call before the break. If, however, a different function call is chosen on the BT menu, revert will go back to the start of this function and open a new break window. The items on the stack above the new starting place will no longer exist. This is used in the tutorial example (see the Break Package Example section above).

`OK` Continue execution from the point of the break. This is useful if you have a simple error, i.e., an unbound variable or a nonnumeric argument to an arithmetic function. Reset the variable in the break window, then select OK. (see the Break Package Example section above).

In addition to being available on the middle button menu of the break window, all of these functions can be typed directly into the window. Only BT behaves differently when typed. It types the stack into the trace window instead of opening a new window.)

Returning to Top Level

Typing Control-D will immediately take you to the top level from any break window. The functions called before the break will stop, but any side effects of the function that occurred before the break remain. For example, if a function set a global variable before it broke, the variable will still be set after typing Control-D.

11. WHAT TO DO IF ...

The purpose of this chapter is to explain what to do in some of the problems commonly experienced by Medley users.

Executive Window turns black

An example is shown in Figure 11-1.

Press any key to unfreeze the window and continue. This pause happens when the command you just typed causes enough information to be printed to fill the window. It gives you a chance to read that one window of text before moving on.

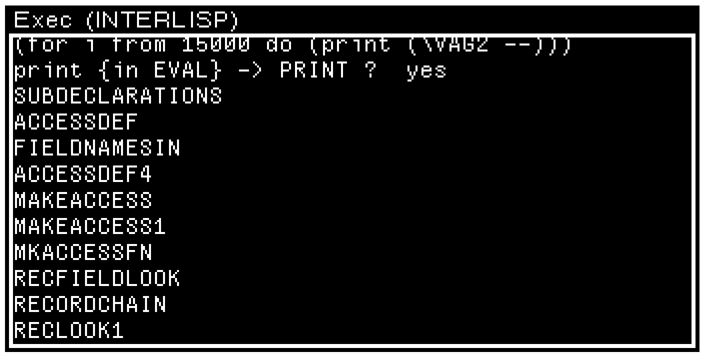


Figure 11-1. Blackened Executive Window

You closed the Executive Window

Open another from the Background Menu.

Mouse disappears

Type **(CURSOR T)** in the Executive Window. The cursor will reappear.

Second window appears

This probably happens because you made a typing mistake, as in Figure 11-2.

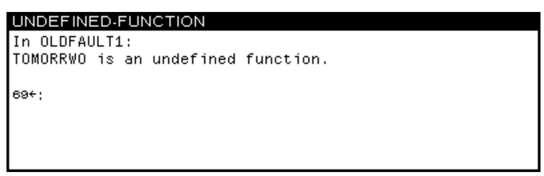


Figure 11-2. Second Window Appears (Break Window) after Typing Error Made

Type a Control-D by simultaneously pressing the Control key and the "D". This aborts the error condition, returning control to the Executive Window.

You keep getting beeped at

Usually the beeping means that Medley want input from you. Look for the flashing caret. It will usually be preceeded by some kind of prompt, indicating what you should type.

You cannot delete the first letter

of the filename you are typing to (FILES?). Type Control-E (error) You will get a linefeed and ←←← printed to the window. Now type the correct filename.

Your function is just sitting there

It is not returning a value, and you think that your program may be in an infinite loop or is having some other major problem. You can see what process is currently running by typing Control-T, or you could interrupt the process by typing Control-E.

A Break Window appears

If the Break Window look something like that shown in Figure 11-3, you are trying to save a file, but there is not enough space on the hard disk.

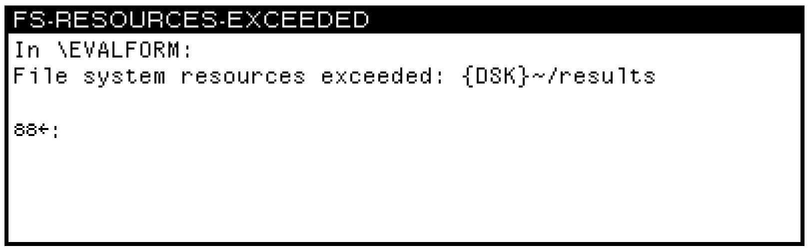


Figure 11-3. Break Window Caused by Insufficient Space in Save File

Exit from the Break Window by typing an up arrow ↑ followed by a Return. Delete old versions of files, and any other files you do not need. Then try again to save the file

You have run out of space

Generally, a Break Window has appeared. The GAINSPACE function allows you to delete non-essential data structures. To use it, type:

(GAINSPACE)

into the Executive Window. Answer N to all questions except the following:

- Delete edit history
- Delete history list
- Delete values of old variables
- Delete your MASTERSCOPE database
- Delete information for undoing your greeting.

Save your work and reload Lisp as soon as possible.

A redefined message appears

The message (*Some.Crucial.Function.Or.Variable* redefined) appears in the Executive Window (see Figure 11-4). The function, variable, or other property has been "smashed" (i.e., its original definition has been changed). If this is not what you wanted, type **UNDO** immediately!

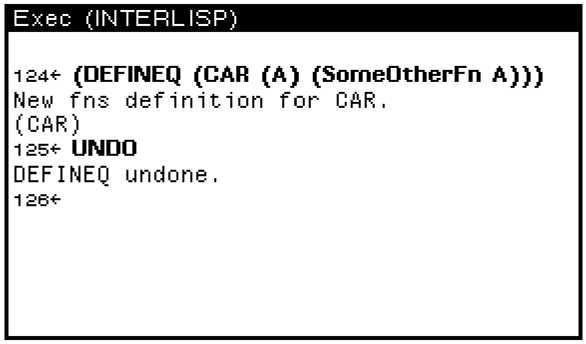


Figure 11-4. CAR redefined!

UNBOUND ATOM

If this occurs, you probably just typed something wrong, or you passed an argument that should have been quoted to a function.

UNDEFINED CAR OF FORM

First, look at what caused the error. If the CAR of the form is a list, then you typed something wrong. If it is an atom, then perhaps that atom does not have a function associated with it. If it is a CLISP word like if or for, then DWIM may have been turned off (see Chapter 9). Type **(DWIM 'C)** to reenable DWIM.

You have traced APPLY

and your screen is spewing out information about everything going on in the environment. Type Control E, and type **(UNBREAK 'APPLY)** before reeturning to the Executive.

[This page intentionally left blank]

12. WINDOWS AND REGIONS

Windows

Windows have two basic parts: an area on the screen containing a collection of pixels, and a property list. The window properties determine how the window looks, the menus that can be accessed from it, what should happen when the mouse is inside the window and a mouse button is pressed, and soon.

CREATEW

Some of the window's properties can be specified when a window is created with the function CREATEW. In particular, it is easy to specify the size and position of the window; its title; and the width of its borders.

(CREATEW *region title borderw'idth*)

Region is a record (named REGION, with the fields `left`, `bottom`, `width`, and `height`) or a list. A region describes a rectangular area on the screen, the window's dimensions and position. The fields left and bottom refer to the position of the bottom leff corner of the region on the screen. Wi dth and height refer to the width and height of the region. The usable space inside the window will be smaller than the width and height, because some of the window's region is consumed by the title bar, and some is taken by the borders.

Title is a string that will be placed in the title bar of the window.

Borderwidth is the width of the border around the exterior of the window, in number of pixels.

For example, typing:

```
(SETQ MY.WINDOW (CREATEW
     (CREATEREGION l00 150 300 200)
     "THIS IS MY OWN WINDOW")
```

or

```
(SETQ MY.WINDOW (CREATEW
      (CREATEW '(100 150 300 200)
     "THIS IS MY OWN WINDOW")
```

produces a window with a default borderwidth. Note that you did not need to specify all the window's properties (see Figure 12-1).

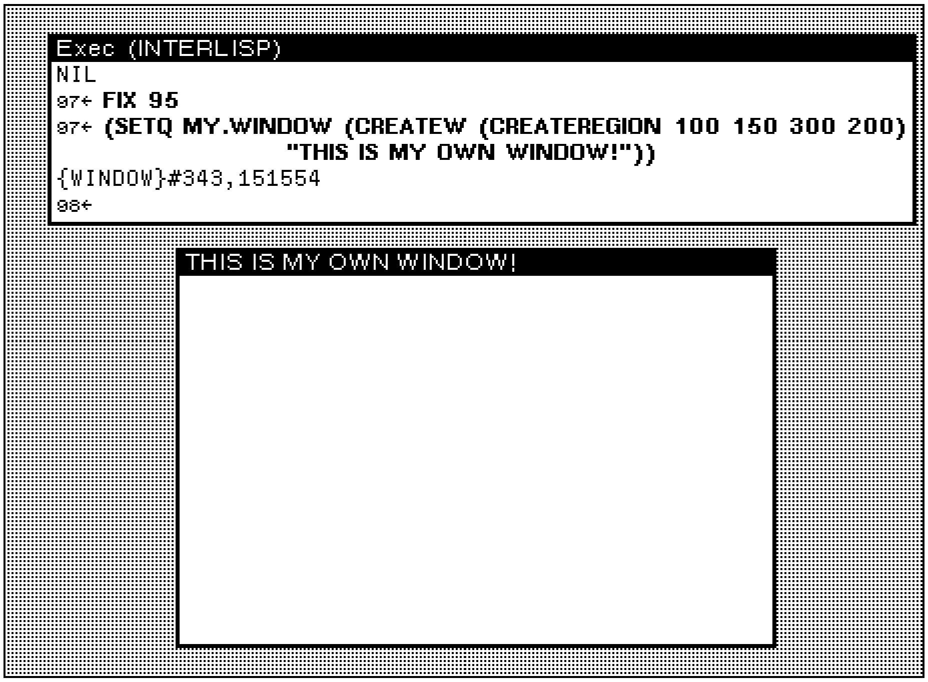


Figure 12-1. Creating a Window

In fact, if `(CREATEW)` is called without specifying a region, you will be prompted to sweep out a region for the window (see Chapter 10)

WINDOWPROP

The function to access or add to any property of a window's property list is `WINDOWPROP`.

`(WINDOWPROP` *window property* <value>`)`

When you use `WINDOWPROP` with only two arguments—window and property—it returns the value of the window's property. When you use `WINDOWPROP` with all three arguments—window, property and value—it sets the value the window's property to the value you inserted for the third argument.

For example, consider the window, `MY WINDOW`, created using `(CREATEW)`. `TITLE` and `REGION` are both properties. Type

```
(WINDOWPROP MY.WINDOW 'TITLE)
```

and the value of `MY.WlNDOW`'s `TITLE` property is returned, `"THIS 1S MY OWN WINDOW"`. To change the title, use the `WINDOWPROP` function, and give it the window, the property title, and the new title of the window.

```
(WINDOWPROP MY.WINDOW 'TITLE "MY FIRST WINDOW")
```

automatically changes the title and automatically updates the window. Now the window looks like Figure 12-2.

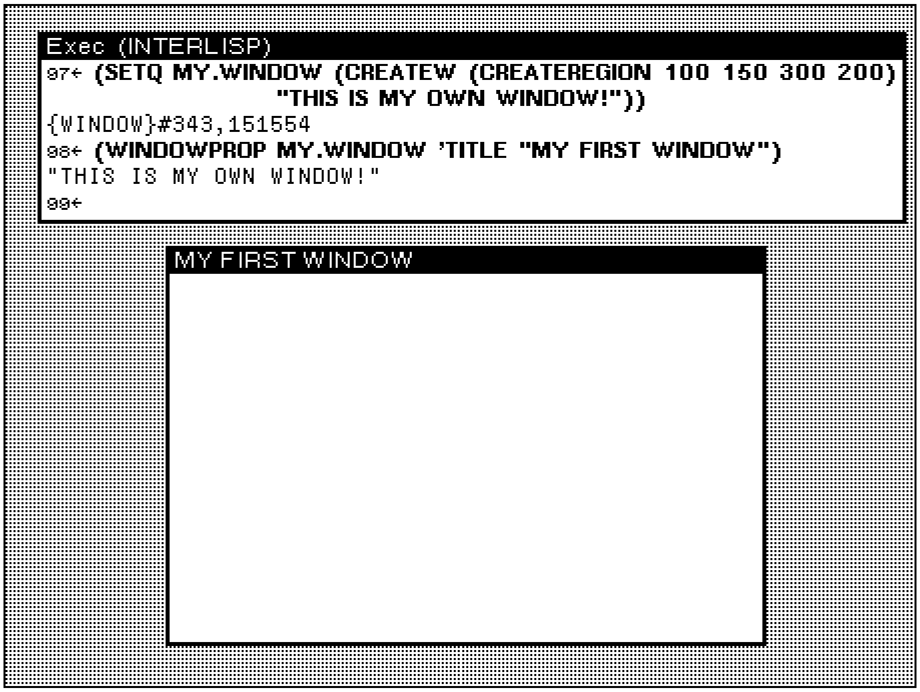


Figure 12-2. TITLE is a Window Property

Altering the region of the window, MY.WINDOW, is also be done with WINDOWPROP, in the same way you changed the title. (Changing either of the first two numbers of a region changes the position of the window on the screen. Changing either of the last two numbers changes the dimensions of the window itself.)

Getting Windows to Do Things

Four basic window properties will be discussed here: CURSORINFN, CURSOROUTFN, CURSORMOVEDFN, and BUTTONEVENTFN.

A function can be stored as the value of the CURSOR1NFN property of a window. It is called when the mouse cursor is moved into that window.

Look at the following example:

1. First, create a window called MY.WINDOW. Type:

```
(SETQ MY.WINDQW
     (CREATEW
        (CREATEREGION 200 200 200 200)
        "THIS WINDOW WILL SCREAM!"))
```

This creates a window.

2. Now define the function SCREAMER. It will be stored on the property CURSOR1NFN. (Notice that this function has one argument, W1NDOWNAME. All functions called from the property CURSOR1NFN are passed the window it was called from. So the value of MY.WINDOW is bound to WINDOWNAME. When it is called, SCREAMER simply rings bells.

```
(DEFINEQ (SCREAMER (WINDOWNAME)
   (RINGBELLS)
   (PROMPTPRINT "YAY - IT WORKS!")
   (RINGBELLS)))
```

3. Now, alter that window's CURSORINFN property, so that the system calls the function SCREAMER at the appropriate time. Type:

```
(WINDOWPROP MY.WINDOW 'CURSORINFN
     (FUNCTION SCREAMER))
```

4. After this, when you move the mouse cursor into MY.WlNDOW, the CURSORINFN property's function is called, and it rings bells twice.

CURSORINFN is one of the many window properties that come with each window - just as REGION and TITLE did. Other properties include:

CURSOROUTFN — The function that is the value of this property is executed when the cursor is moved out of a window.

CURSORMOVEDFN — The function that is the value of this property is executed when the cursor is moved while it is inside the window.

BUTTONEVENTFN — The function that is the value of this property is executed when either the left or middle mouse buttons are pressed (or released).

Figure 12-3 shows MY.WlNDOW's properties. Notice that the CURSORINFN has the function SCREAMER stored in it. The properties were shown in this window using the function INSPECT. INSPECT is covered in Chapter 17.

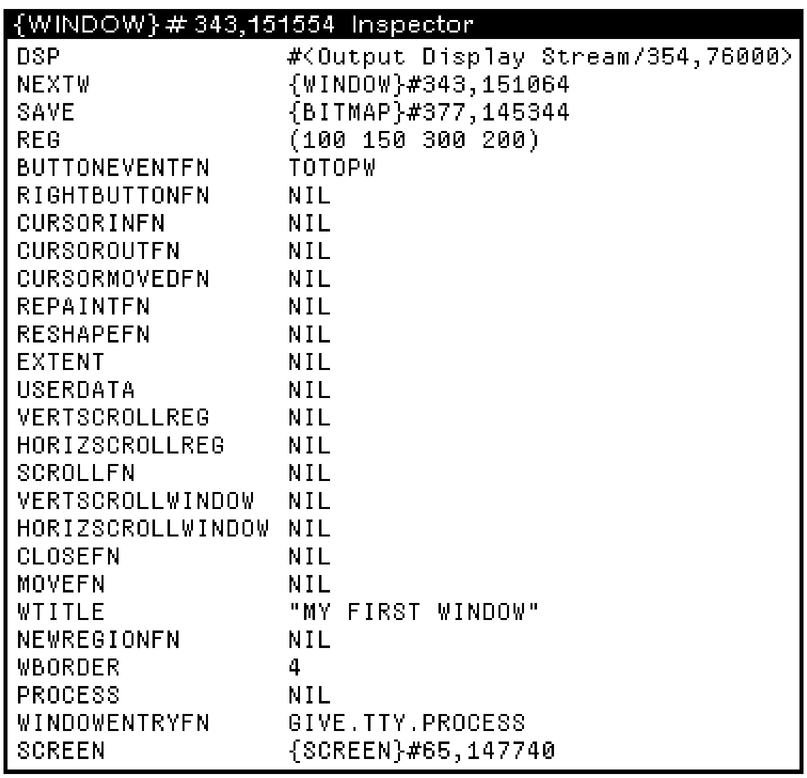


Figure 12-3. Inspecting MY.WINDOW for Mouse-Related Window Properties

You can define functions for the values of the properties CURSOROUTFN and CURSORMOVEDFN in much the same way as you did for CURSORINFN. The function that is the value of the property BUTTONEVENTFN, however, can be specialized to respond in different ways, depending on which mouse button is pressed. This is explained in the next section.

BUTTONEVENTFN

BUTTONEVENTFN is another property of a window. The function that is stored as the value of this property is called when tho mouse is inside the window, and a mouse button is pressed. As an example of how to use it, type:

```
(WINDOWPROP MY.WINDOW 'BUTTONEVENTFN
     (FUNCTION SCREAMER))
```

When the mouse cursor is moved into the window, bells will ring because of the CURS0R1NFN, but it will also ring bells when either the left or middle mouse button is pressed. Notice that the right mouse button functions as it usually does, with the window manipulation menu. If only the left button should evoke the function SCREAMER, then the function can be written to do just this, using the function MOUSESTATE, and a form that only MOUSESTATE understands, ONLY. For example:

```
(DEFINEQ
     (SCREAMER2 (WINDOWNAME)
          (if (MOUSESTATE (ONLY LEFT))
               then (RINGBELLS))))
```

In addition to (ONLY LEFT), MOUSESTATE can also be passed (ONLY MIDDLE), (ONLY RIGHT) or combinations of these (e.g. (OR (ONLY LEFT) (ONLY MIDDLE))). You do not need to use ONLY with MOUSESTATE for every application. ONLY means that that button is pressed and no other.

If you do write a function using (ONLY RIGHT), be sure that your function also checks position of the mouse cursor. Even if you want your function to be executed when the mouse cursor is inside the window and the right button is pressed, there is a convention that the function DOWINDOWCOM should be executed when the mouse cursor is in the title bar or the border of the window and the right mouse button is pressed. Please program your windows using this tradition! For more information, please see Chapter 28 in the *IRM*.

Looking at a Window's Properties

INSPECT is a function that displays a list of the properties of a window, and their values. Figure 12.3 shows the INSPECT function run with MY.WINDOW. Note the properties introduced in CREATEW: WBORDER is the window's border, REG is the region, and WTITLE is the window's title.

Regions

A region is a record, with the fields LEFT, BOTTOM, WIDTH, and HEIGHT. LEFT and BOTTOM refer to where the bottom left hand corner of the region is positioned on the screen. WIDTH and HEIGHT refer to the width and height of the region.

CREATEREGION creates an instance of a record of type REGION. Type:

```
(SETQ MY.REGION (CREATEREGION 15 l00 200 450))
```

to create a record of type REGION that denotes a rectangle 200 pixels high, and 450 pixels wide, whose bottom left corner is at position (15, 100). This record instance can be passed to any function that requires a region as an argument, such as CREATEW, above.

13. WHAT ARE MENUS?

While Medley provides a number of menus of its own (see Chapter 3), this section addresses the menus you wish to create. You will learn how to create a menu, display a menu, and define functions that make your menu useful. Menus are instances of records (see Chapter 24). There are 27 fields that determine the composition of every menu. Because Medley provides default values for most of these descriptive fields, you need to familiarize yourself with only a few that we describe in this section.

Two of these fields, the `TITLE` of your menu, and the `ITEMS` you wish it to contain, can be typed into the executive window as shown below:

```
Exec 2 (INTERLISP)

2/154← (SETQ MY.MENU (CREATE MENU
        TITLE ← "PLEASE CHOOSE ONE OF THE
ITEMS"
        ITEMS ← '(QUIT NEXT-QUESTION
                  NEXT-TOPIC SEE-TOPICS)))
{MENU}#374,123464
2/155←
```

Figure 13-1. Creating a menu

Note that creating a menu does not display it. `MY.MENU` is set to an instance of a menu record that specifies how the menu will look, but the menu is not displayed.

Displaying Menus

Typing either the `MENU` or `ADDNENU` functions will display your menu on the screen. `MENU` implements pop-up menus, like the Background Menu or the Window Menu. `ADDMENU` puts menus into a semi-permanent window on the screen, and lets you select items from it.

`(MENU MENU POSITION)` pops up a menu at a particular position on the screen.

Type:

```
(MENU MY.MENU NIL)
```

to position the menu at the end of the mouse cursor. Note that the `POSITION` argument is `NIL`. In order to go on, you must either choose an item, or move outside the menu window and press a mouse button. When you do either, the menu will disappear. If you choose an item, then want to choose another, the menu must be redisplayed.

`(ADDMENU` *menu window position*`)` positions a permanent menu on the screen, or in an existing window.

Type:

```
(ADDMENU MY.MENU)
```

to display the menu as shown in Figure 13-2. This menu will remain active, (will stay on the screen) without stopping all the other processes. Because `ADDMENU` can display a menu without stopping all other processes, it is very popular in users programs.

If window is specified, the menu is displayed in that window. If window is not specified, a window the correct size for the menu is created, and the menu is displayed in that window.

If position is not specified, the menu appears at the current position of the mouse cursor.

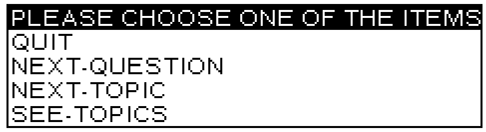


Figure 13-2. Simple MenuDisplayed with ADDMENU

Getting Menus to Do Stuff

One way to make a menu do things is to specify more about the menu items. Instead of items simply being the strings or atoms that will appear in the menu, items can be lists, each list with three elements (see Figure 13-3). The first element of each list is what will appear in the menu; the second expression is what is evaluated, and the results of the evaluation returned, when the item is selected; and the third expression is the expression that should be printed in the Prompt window when a mouse button is held down while the mouse is pointing to that menu item. This third item should be thought of as help text for the user. If the third element of the list is NIL, the system responds with **Will select this item when you release the button**.

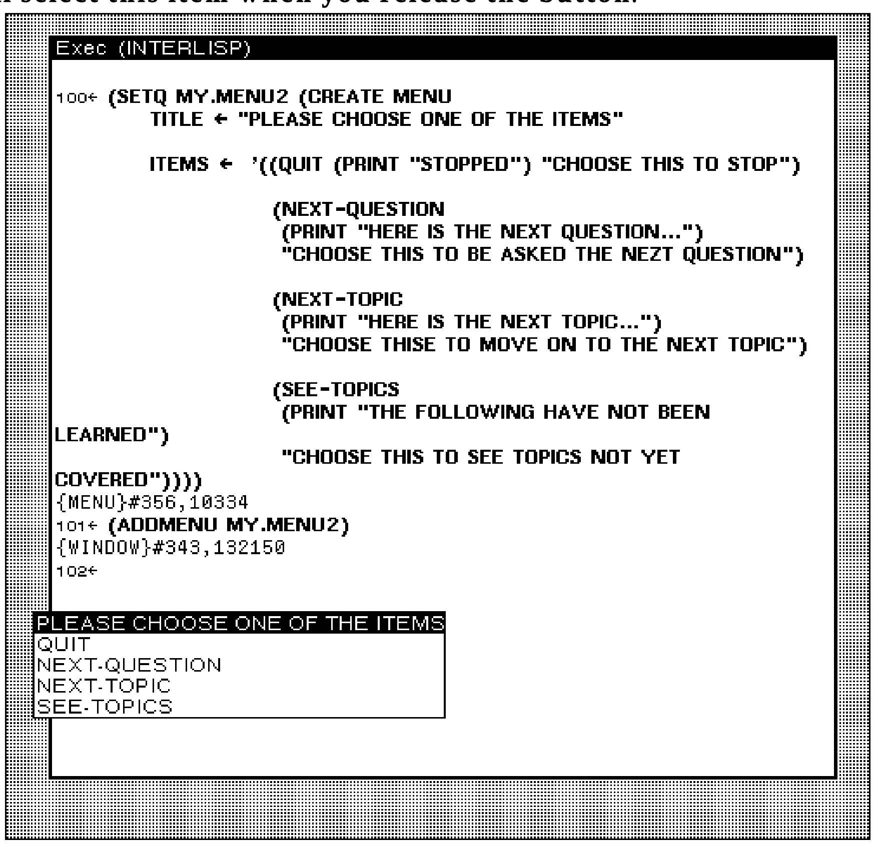


Figure 13-3. Creating a Menu to do Things, then displaying it with the function ADDMENU

Now when an item is selected from MY.MENU2, something will happen. When a mouse button is held down, the expression typed as the third element in the item's specification will be printed in the Prompt window. (See Figure 13-4.)

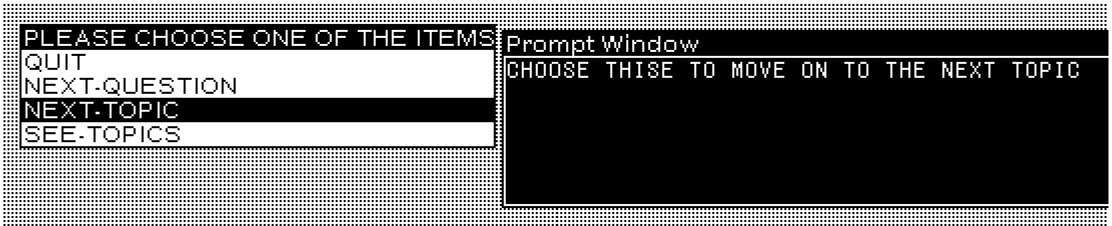


Figure 13-4. Mouse Button Held Down While Mouse Cursor SeIects NEXT.QUESTION

When the mouse button is released (i.e., the item is selected) the expression that was typed as the second element of the item's specification will be run. (See Figure 13-5.)

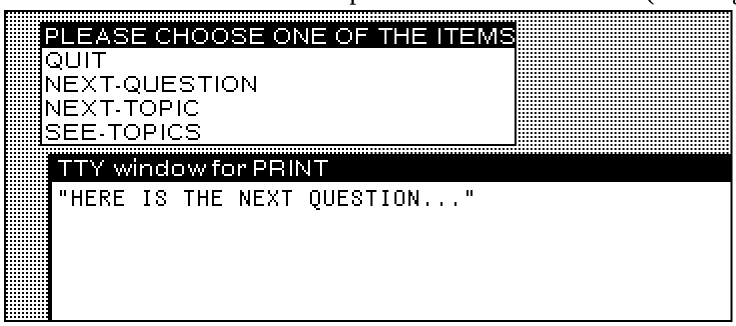


Figure 13-5. NEXT-QUESTION Selected

WHENHELDFN and WHENSELECTEDFN Fields of a Menu

Another way to get a menu to do things is to define functions, and make them the values of the menu's WHENHELDFN and WHENSELECTEDFN fields. As the value of the WHENHELDFN field of a menu, the function you defined will be executed when you press and hold a mouse button inside the menu. As the value of the WHENSELECTEDFN field of a menu, the function you defined will be executed when you choose a menu item. This example has the same functionality as the previous example, where each menu item was entered as a list of three items.

As an example, type in these two functions so that they can be executed when the menu is created and displayed:

```
(DEFINEQ (MY.MENU3.WHENHELD (ITEM.SELECTED MENU.FROM
BUTTON.PRESSED)
(SELECTQ ITEM.SELECTED
      (QUIT (PROMPTPRINT "CHOOSE THIS TO STOP")
      (NEXT-QUESTION (PROMPTPRINT "CHOOSE THIS TO BE ASKED THE
            NEXT QUESTION"))
      (NEXT-TOPIC (PROMPTPRINT "CHOOSE THIS TO MOVE ON TO THE
            NEXT SUBJECT"))
      (SEE-TOPICS (PROMPTPRINT "CHOOSE THIS TO SEE THE TOPICS
            NOT YET LEARNED"))
      (ERROR (PROMPTPRINT "NO MATCH FOUND"))))
```

```
(DEFINEQ (MY.MENU3.WHENSELECTED (ITEM.SELECTED MENU.FROM
BUTTON.PRESSED)
(SELECTQ ITEM.SELECTED
      (QUIT (PRINT "STOPPED")
      (NEXT-QUESTION (PRINT "HERE IS THE NEXT QUESTION"))
      (NEXT-TOPIC (PRINT "HERE IS THE NEXT SUBJECT"))
      (SEE-TOPICS (PRINT "THE FOLLOWING HAVE NOT BEEN
            LEARNED . . ."))
      (ERROR (PROMPTPRINT "NO MATCH FOUND"))))
```

Now, to create the menu, type:

```
(SETQ MY.MENU3 (CREATE MENU
    TITLE← "PLEASE CHOOSE ONE OF THE ITEMS"
    ITEMS ← '(QUIT NEXT-QUESTION NEXT-TOPIC SEE-TOPICS)
    WHENHELDFN ← (FUNCTION MY.MENU3.WHENHELD)
    WHENSELECTEDFN ← (FUNCTION MY.MENU3.WHENSELECTED)))
```

To see your menu work, type

```
(ADDMENU MY.MENU3)
```

Now, due to executing the WHENHELDFN function, holding down any mouse button while pointing to a menu item will display an explanation of the item in the prompt window. The screen will once again look like Figure 13-4 when the mouse button is held when the mouse cursor is pointing to the item NEXT-TOPIC.

Now due to executing the WHENSELECTEDFN function, releasing the mouse button to select an item will cause the proper actions for that item to be taken. The screen will once again look like Figure 13-5 when the item NEXT-TOPIC is selected. The crucial thing to note is that the functions you defined for WHENHELDFN and WHENSELECTEDFN are automatically given the following arguments:

1. The item that was sølected, ITEM.SELECTED
2. The menu it was selected from, MENU.FROM
3. The mousø button that was pressed BUTTON PRESSED

These functions, MY.MENU3.WHENHELD and MY.MENU3.WHENSELECTED, were quoted using FUNCTION instead of QUOTE both for program readability and so that the compiler can produce faster code when the program is compiled. It is good style to quote functions in Lisp by using the function FUNCTION instead of QUOTE.

Looking at a Menu's Fields

INSPECT is a function that displays a list of the fields of a menu, and their values. Figure 13-6 shows the various fields of MY.MENU3 when the function (INSPECT MY.MENU3) was called. Notice the values that were assigned by the examples, and all the defaults.

```
161← (INSPECT MY.MENU3)
{WINDOW}#357,73064
162←
{MENU}# 336,174464 Inspector
ITEMWIDTH          236
ITEMHEIGHT         12
IMAGEWIDTH         238
IMAGEHEIGHT        62
MENUREGIONLEFT     -1
MENUREGIONBOTTOM   -1
IMAGE              {WINDOW}#376,26000
SAVEIMAGE          NIL
ITEMS              (QUIT NEXT-QUESTION NEXT-TOPIC SEE-T
MENUROWS           4
MENUCOLUMNS        1
MENUGRID           (0 0 236 12)
CENTERFLG          NIL
CHANGEOFFSETFLG    NIL
MENUFONT           {FONTDESCRIPTOR}#74,70204
TITLE              "PLEASE CHOOSE ONE OF THE ITEMS"
MENUOFFSET         (0 . 0)
WHENSELECTEDFN     MY.MENU3.WHENSELECTED
MENUBORDERSIZE     0
MENUOUTLINESIZE    1
WHENHELDFN         MY.MENU3.WHENHELD
MENUPOSITION       NIL
WHENUNHELDFN       CLRPROMPT
MENUUSERDATA       NIL
MENUTITLEFONT      NIL
SUBITEMFN          NIL
MENUFEEDBACKFLG    NIL
SHADEDITEMS        NIL
```

Figure 13-6. MY.MENU3 Fields

14. BITMAPS

A bitmap is a rectangular array of dots. The dots are called "pixels" (for picture elements). Each dot, or pixel, is represented by a single bit. When a pixel or bit is turned on (i.e. that bit set to 1), a black dot is inserted into a bitmap. If you have a bitmap of a floppy on your screen (Figure 14-1), then all of the bits in the area that make up the floppy are turned on, and the surrounding bits are turned off.

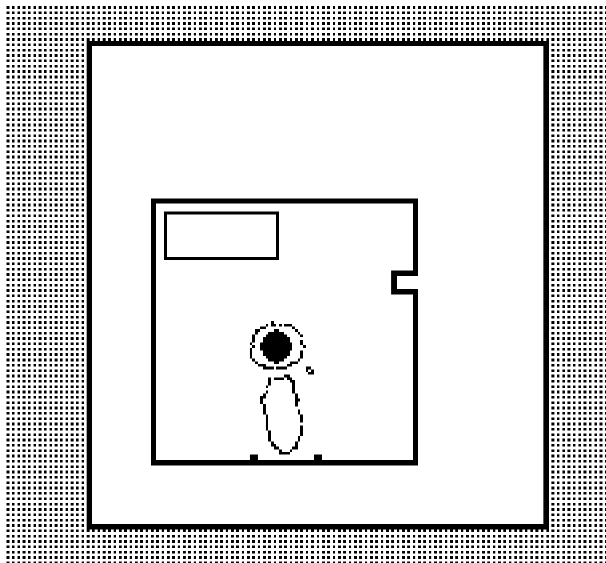


Figure 14-1. Bitmap of a Floppy

`BITMAPCREATE` creates a bitmap, even though it can't be seen.

(BITMAPCREATE *width height*)

If the width and height are not supplied, the system will prompt you for them.

`EDITBM` edits the bitmap. The syntax of the function is:

(EDITBM *bitmapname*)

Try the following to produce the results in Figure 14-4:

```
(SETQ MY.BITMAP (BITMAPCREATE 60 40))
EDITBM MY.BITMAP)
```

To draw In the bitmap, move the mouse into the gridded section of the bitmap editor, and press and hold the leff mouse button. Move the mouse around to turn on the bits represented by the spaces in the grid. Notice that each space in the grid represents one pixel on the bitmap

To erase Move the mouse into the gridded section of the bitmap editor, and press and hold the center mouse button. Move the mouse around to turn off the bits represented by the spaces in the gridded section of the bitmap editor.

To work on a different section Point with the mouse cursor to the picture of the actual bitmap (the upper left corner of the bitmap editor). Press and hold the left mouse button. A menu with the single item, Move will appear. (See Figure 14-2.) Choose this item.

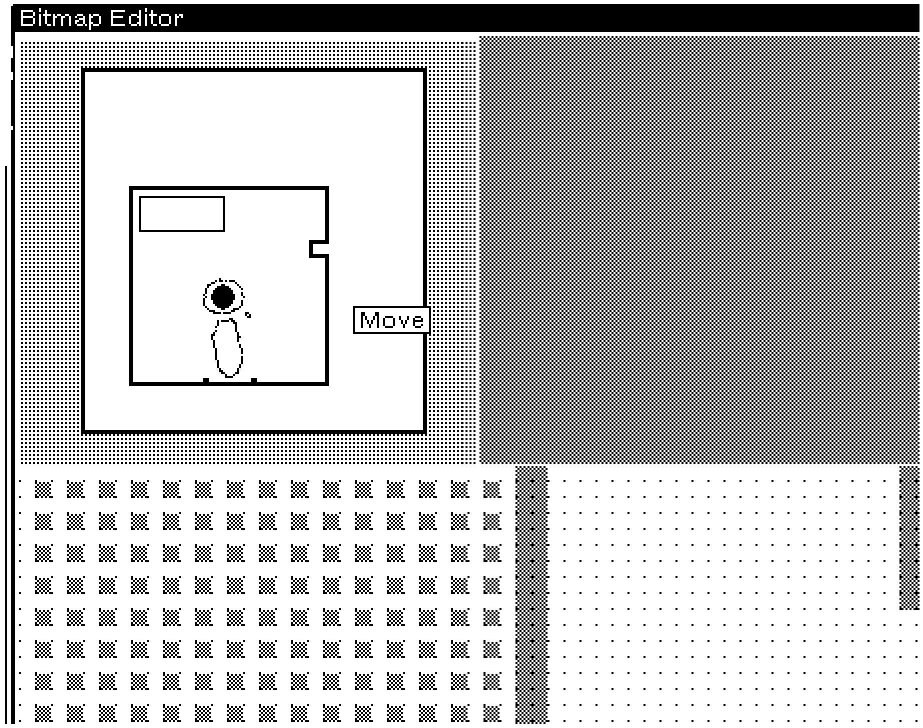


Figure 14-2. Menu with Single Item (Move)

You will be asked to position a ghost window over the bitmap. This ghost window represents the portion of the bitmap that you are currently editing. Place it over the section of the bitmap that you wish to edit and click the left mouse button (see Figure 14-3).

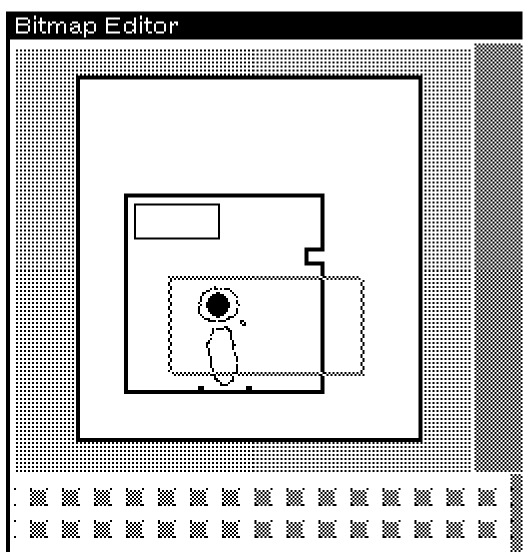


Figure 14-3. Ghost Window Awaiting Positioning

To end the session, bring the mouse cursor into the upper-right portion of the window (the grey area) and press the center button. Select OK from the menu to save your artwork.

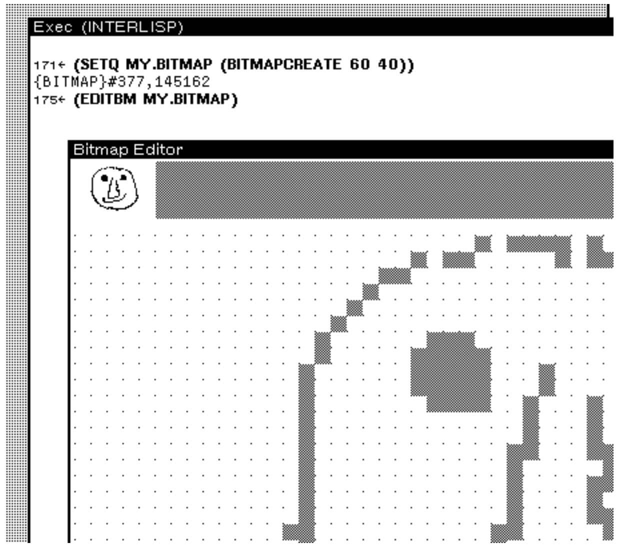


Figure 14-4. Editing a Bitmap

BITBLT is the primitive function for moving bits (or pixels) from one bitmap to another. It extracts bits from the source bitmap, and combines them in appropriate ways with those of the destination bitmap. The syntax of the function is:

```
(BITBLT sourcebitmap sourcelefl sourcebottom destinationbitmap
destinationleft destinationbottom width height sourcetype
operation texture clippIngregion)
```

Here's how it's done —using MY.BITMAP as the sourcebitmap and MY.WlNDOW as the destinationbitmap.

```
(BITBLT MY.BITMAP NIL NIL
        MY.WINDOW NIL NIL NIL NIL 'INPUT 'REPLACE)
```

Note that the destination bitmap can be, and usually is, a window. Actually, it is the bitmap of a window, but the system handles that detail for you. Because of the NILs (meaning "use the default"), MY.BITMAP will be BITBLT'd into the lower right corner of MY.WlNDOW (see Figure 14-5).

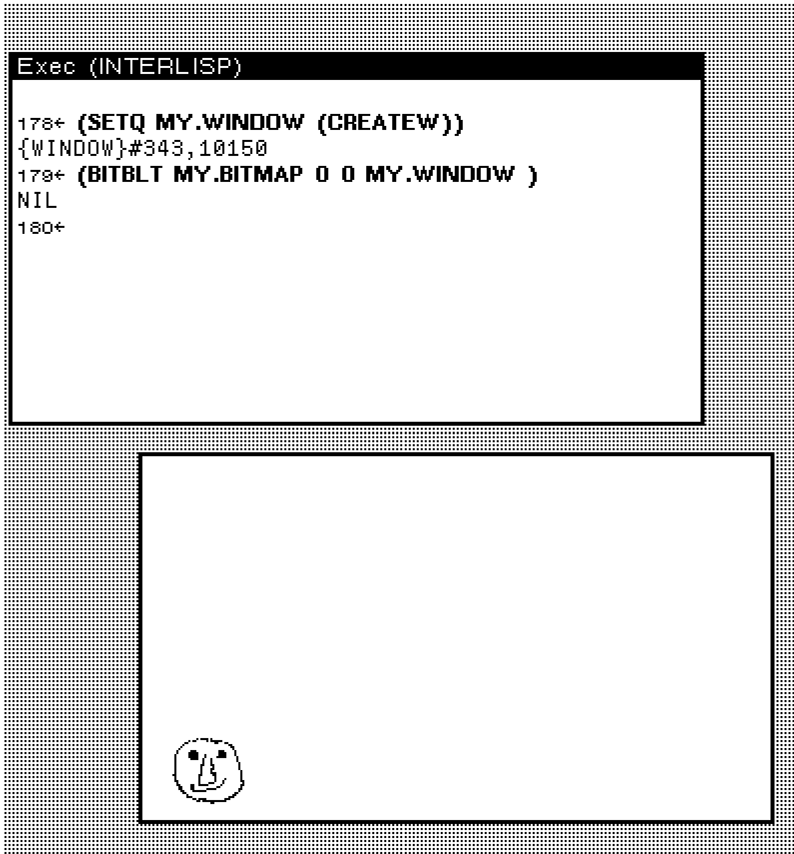


Figure 14-5. BITBLTing a Bitmap onto a Window

Here is what each of the BITBLT arguments to the function mean:

sourcebitmap	The bitmap to be moved into the destinationbitmap
sourceleft	A number, starting at 0 for the left edge of the sourcebitmap, that tells BITBLT where to start moving pixels from the sourcebitmap. For example, if the leftmost 10 pixels of sourcebitmap were not to be moved, sourceleft should be 10. The default value is 0.
sourcebottom	A number, starting at 0 for the bottom edge of the sourcebitmap, that tells BITBLT where to start moving pixels from the sourcebitmap. For example, if the bottom 10 rows of pixels of sourcebitmap were not to be moved, sourcebottom should be 10 The default value is 0.
destinationbitmap	The bitmap that will receive the sourcebitmap. This is often a window (actually the bitmap of a window, but Interlisp-D takes care of that for you).
destinationleft	A number, starting at 0 for the left edge of the destinationbitmap, that tells BITBLT where to start placing pixels from the sourcebitmap. For example, to place the sourcebitmap 10 pixels in from the left, destinationleft should be 10. The default value is 0.

`destinationbottom`	A number, starting at 0 for the bottom edge of the destinationbitmap, that tells `BITBLT` where to start placing pixels from the sourcebitmap. For example, to place the sourcebitmap 10 pixels up from the bottom, destinationbottom should be 10. The default value is 0.
`width`	How many pixels in each row of sourcebitmap should be moved. The samc amount of space is used in destinationbitmap to receive the sourcebitmap. If this argument is `NIL`, it defaults to the number of pixels from sourceleft to the end of the row of sourcebitmap.
`height`	How many rows of pixels of sourcebitmap should be moved. The same amount of space is used in destinationbitmap to receive the sourcebitmap. If this argument is `NIL`, it defaults to the number of rows from sourcebottom to the top of the sourcebitmap.
`sourcetype`	Refers to one of three ways to convert the sourcebitmap for writing. For now, just use `'INPUT`.
`operation`	Refers to how the sourtebitmap gets `BITBLT`'d on to the destinationbitmap. `'REPLACE` will `BLT` the exact sourcebitmap. Other operations allow you to `AND`, `OR` or `XOR` the bits from the sourcebitmap onto the bits on the destinationbitmap.
`texture`	Just use `NIL` for now.
`clippingregion`	Just use `NIL` for now.

For more information on these operations, see Chapter 27 in the *IRM*.

15. DISPLAYSTREAMS

A displaystream is a generaJized "place to display". They determine exactly what is displayed where. One example of a displaystream is a window. Windows are the only displaystreams that will be used in this chapter. If you want to draw on a bitmap that is not a window, other than with BITBLT, or want to use other types of displaystreams, please refer to Chapter 27 in the *IRM*.

This chapter explains functions for drawing on displaystreams: DRAWLINE, DRAWTO, DRAWCIRCLE., and FILLCIRCLE. In addition, functions for locating and changIng your curreAt position in the displaystream are covered: DSPXPOSITION, DSPYPOSITION, and MOVETO.

Drawing on a Displaystream

The examples belowshow you how the functions for drawing on a display stream work. First, create a window. Windows are displaystreams, and the one you create are used for the examples in this chapter. Type:

```
(SETQ EXAMPLE.WINDOW (CREATEW))
```

DRAWLINE

DRAWLINE draws a line in a displaystream. For example, type:

```
(DRAWLINE 10 15 100 150 5 'INVERT EXAMPLE.WINDOW)
```

The results should look like Figure 15-1:

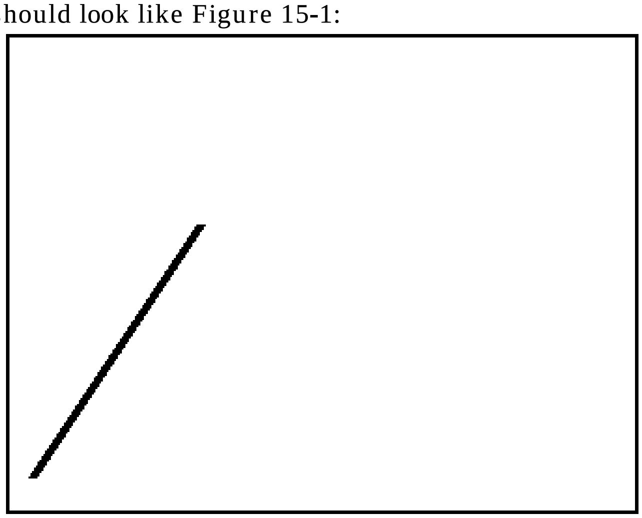


Figure 15-1. Line Drawn onto the EXAMPLE.WINDOW Displayrtream

The syntax of DRAWLINE is

(DRAWLINE *x1 y1 x2 y2 width operation stream color dashing*)

The coordinates of the left bottom corner of the displaystream are 0 0.

xl and yl	x and y coordinates of the beginning of the line
x2andy2	ending coordinates of the line
width	width of the line, in pixels
operation	way the line is to be drawn. `INVERT` causes the line to invert the bits that are already in the displaystream. Drawing a line the second time using `INVERT` erases the line. For other operations, see Chapter 27 in the *IRM*.
stream	displaystream. In this case, you used a window.

DRAWTO

`DRAWTO` draws a line that begins at your current position in the displaystream. For example, type:

```
(DRAWTO 120 135 5 'INVERT EXAMPLE.WINDOW)
```

The results should look like Figure 15-2:

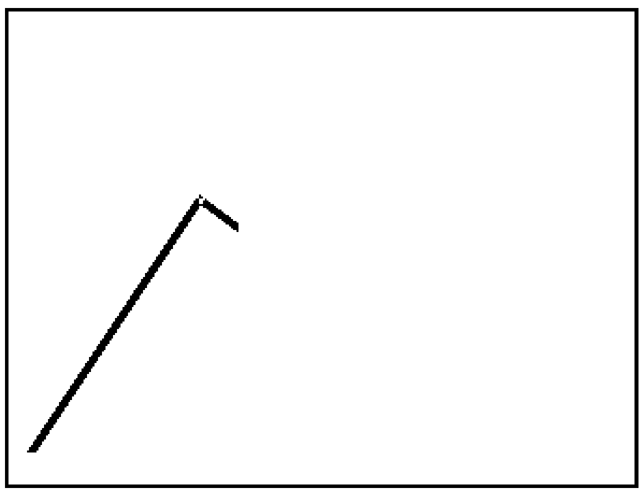


Figure 15-2. Another Line drawn onto the `EXAMPLE.WINDOW` Displaystream

The syntax of `DRAWTO` is

(`DRAWTO` *x y width operation stream color dashing*)

The line begins at the current position in the displaystream.

x	x coordinate of the end of the line
y	y coordinate of the end of the line
width	width of the line
operation	way the lino is to be drawn. `INVERT` causes the line to invert the bits that aro already in tho displaystream. Drawing a line the second time using `INVERT` erases the line. For other operations, see Chapter 27 in the *IRM*
stream	displaystreom. In this case. you used a window.

DRAWCIRCLE

DRAWCIRCLE draws a circle on a displaystream. To use it, type:

```
(DRAWCIRCLE 150 100 30 '(VERTICAL 5) NIL EXAMPLE.WINDOW)
```

Now your window, EXAMPLE.WlNDOW, should look like Figure 15-3:

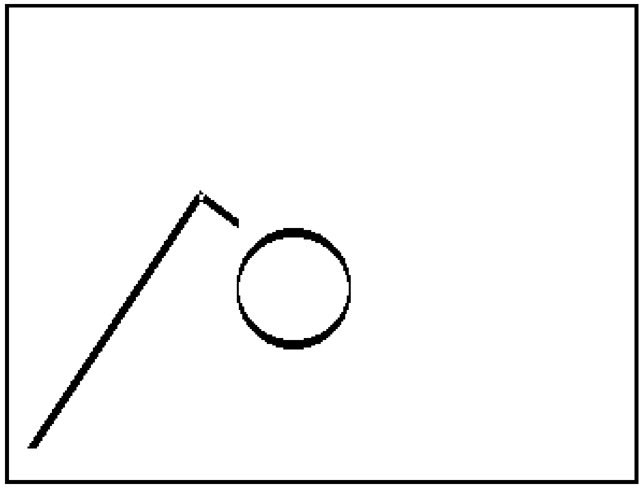


Figure 15-3. Circle Drawn onto the EXAMPLE.WlNDOW Displaystream

The syntax of DRAWCIRCLE is

(DRAWCIRCLE *centerx centery radius brush dashing stream*)

centerx	x coordinate of the center of the circle
centery	coordinate of the center of the circle
radius	radius of the circle in pixels
brush	list.- The first- item of the list is the shape of the brush. Some of your options include ROUND, SQUARE, and VERTICAL. The second item of that list is the width of the brush in pixels.
dashing	list of positive integers. The brush is "on" for the number of units indicated by the first element of the list, "off" for the number of units indicated by the second element of the list. The third element specifies how long it will be on again, and so forth. The sequence is repeated until the circle has been drawn.
stream	displaystream. In this case, you used a window.

FILLCIRCLE

FILLCIRCLE draws a filled circle on a displaystream. To use it, type:

```
(FILLCIRCLE 200 150 10 GRAYSHADE EXAMPLE.WINDOW)
```

EXAMPLE.WlNDOW now looks like Figure 15-4:

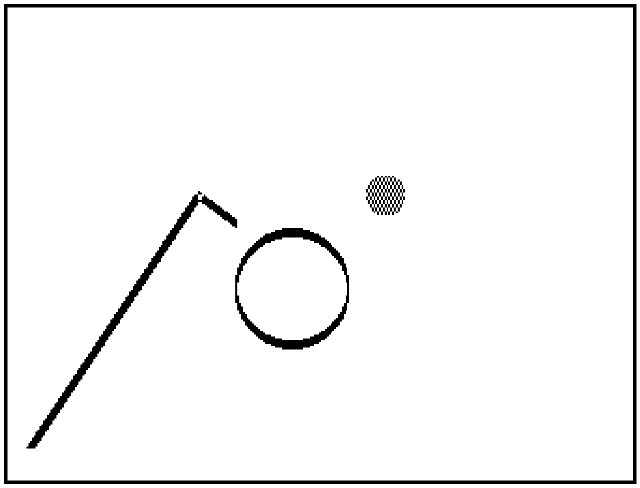


Figure 15-4. A filled circle drawn onto the displaystream

The syntax of FILLCIRCLE is:

(FILLCIRCLE *centerx centery radius texture stream*)

centerx	x coordinate of the center of the circle
centery	y coordinate of the center of the ci rcle
radius	radius of the circle in pixels
texture	shade that will be used to fill in the circle. Interlisp-D provides you with three shades: WHlTESHADE, BLACKSHADE, and GRAYSHADE. You can also create your own shades. For more information on how to do this, see Chapter 27 in the *IRM*.
stream	displaystream. In this case, you used a window

There are many other functions for drawing on a displaystream. Please refer to Chapter 27 in the *IRM*.

Text can also be placed into displaystreams. To do this, use printing functions such as PRIN1 and PRIN2, but supply the name of the displaystream as the "file" to print to. To place the text in the proper position in the displaystream, see the section below.

Locating and Changing Your Position in a Displaystream

There are functions provided to locate, and to change your current position in a displayitream. This can help you place text, and other images where you want them in a displaystream. This primer will only discuss three of these. There are others, and they can be found in the Chapter 27 of the *IRM*.

DSPXPOSITION

DSPXPOSITION is a function that will either change the current x position in a displaystream, or simply report it. To have the function report the current x position in EXAMPLE.WlNDOW, type:

```
(DSPXPOSITION NIL EXAMPLE.WINDOW)
```

`DSPXPOSITION` expects two arguments. The first is the new x position. If this argument is `NIL`, the current position is not changed, merely reported. The second argument is the displaystream.

DSPYPOSITION

`DSPYPOSITION` is an analogous function, but It changes or reports the current y position in a displaystream. As with `DSPXPOSlTlON`, if the first argument is a number, the current y position will be changed to that position. If it is `NIL`, the current position is simply reported. To have the function report the current y position in `EXAMPLE.WlNDOW`, type:

```
(DSPYPOSITION NIL EXAMPLE.WlNDOW)
```

MOVETO

The function `MOVETO` always changes your position in the displaystream. It expects three arguments:

(`MOVETO` *x y stream*)

x	new x position in the display stream
y	new y position in the display stream
stream	display stream. The examples so far have used a window

16. FONTS

This chapter explains fonts and fontdescriptors, what they are and how to use them, so that you can use functions requiring fontdescriptors

You have already been exposed to many fonts in Medley. For example, when you use the structure editor, DEdit (see the Using the List Structure Editor section of Chapter 7), you noticed that the comments were printed in a smaller font than the code, and that CLlSP words (see the CLISP section of Chapter 9) were printed in a darker font than the other words in the function. These are only some of the fonts that are available in Medley.

In addition to the fonts that appear on your screen, Medley uses fonts for printers that are different than the ones used for the screen. The fonts used to print to the screen are called `DlSPLAYFONTS`. The fonts used for prining are called `INTERPRESSFONTS`, or `PRESSFONTS`, depending on the type of printer.

What Makes Up a Font

Fonts are described by family, weight, slope, width, and size. This section discusses each of these, and describes how they affect the font you see on the screen.

Family is one way that fonts can differ. Here are some examples of how "family" affects the look of a font:

`CLASSIC` This family makes the word "Able" look like this: Able

`MODERN` This family makes the word "Able" look like this: Able

`TITAN` This family makes the word "Able" look like this: `Able`

Weight also determines the look of a font. Once again, "Able" will be used as an example, this time only with the Classic family. A font’s weight can be:

`BOLD` And look like this: **Able**

`MEDIUM`
or `REGULAR` And look like this: Able

The slope of a font is italic or regular. Using the Classic family font again, in a regular weight, the slope affects the font like this:

`ITALIC` Looks like this: *Able*

`REGULAR` Looks like this: Able

The width of a font is called its "expansion". It can be `COMPRESSED`, `REGULAR`, or `EXPANDED`.

Together, the weight, slope, and expansion of a font specifies the font’s "face". Specifically, the face of a font is a three element list:

(weight slope expansion)

To make it easier to type, when a function requires a font face as an argument, it can be abbreviated with a three-character atom. The first specifies the weight, the second the slope, and the third character the expansion. For example, some common font faces are abbreviated:

MRR	This is the usual face, MEDIUM, REGULAR, REGULAR
MIR	Makes an italic font. It stands for: MEDIUM, ITALIC, REGULAR
BRR	Makes a bold font. The abbreviation means: BOLD, REGULAR, REGULAR
BIR	Means that the font should be both bold and italic. BIR stands for BOLD, ITALIC, REGULAR

The above examples are used so oflen, that there are also more mnemonic abbreviations for them. They can also be used to specify a font face for a function that requires a face as an argument. They are:

STANDARD	This is the usual face: MEDIUM, REGULAR, REGULAR; it was abbreviated above, MRR
ITALIC	This was abbreviated above as MIR, and specifies an italic font
BOLD	Makes a bold font; it was abbreviated above, BRR
BOLDITALIC	Makes a font both bold and italic: BOLD, ITALIC, REGULAR; it was abbreviated above, BIR

A font also has a size. It is a positive integer that specifies the height of the font in printers points. A point is, on an 1108 screen, about 1/72 of an inch. On the screen of an 1186, a point is 1/80 of an inch. The size of the font used in this chapter is 10. For comparison, here is an example of a TITAN, MRR, size 12 font: Able.

Fontdescriptors and FONTCREATE

For Medley to use a font, it must have a fontdescriptor. A fontdescriptor is a data type in Interlisp-D that that holds all the information needed in order to use a particular font. When you print out a fontdescriptor, it looks like this:

```
{FONTDESCRIPTOR}#74,45540
```

Fontdescriptors are created by the function FONTCREATE. For example,

```
(FONTCREATE 'HELVETICA 12 'BOLD)
```

creates a fontdescriptor that, when used by other functions, prints in HELVETICA BOLD size 12. Interlisp-D functions that work with fonts expect a fontdescriptor produced with the FONTCREATE function.

The syntax of FONTCREATE is:

(FONTCREATE *family size face*)

Remember from the previous section, face is either a three element list (weight slope expansion), a three character atom abbreviation, e.g. MRR, or one of the mnemonic abbreviations, e.g. STANDARD.

If FONTCREATE is asked to create a fontdescriptor that aJready exists, the existing fontdescriptor is simply returned.

Display Fonts

Display fonts require files that contain the bitmaps used to print each character on the screen. All of these files have the extension .DlSPLAYFONT. The file name itself describes the font style and size that uses its bitmaps. For example:

    MODERN12.DISPLAYFONT

contains bitmaps for the font family MODERN in size 12 points. Wherever you put your .DISPLAYFONT files, you should make this one of the values of the variable DISPLAYFONTDIRECTORIES. Its value is a list of directories to search for the bitmap files for display fonts. Usually, it contains the "FONT" directory where you copied the bitmap files, and the current connected directory. The current connected directory is specified by the atom NIL. When looking for a .DISPLAYFONT file, the system checks the FONT directory on the hard disk, then the current connected directory.

Figure 16-1 shows an example value of DISPLAYFONTDIRECTORIES:

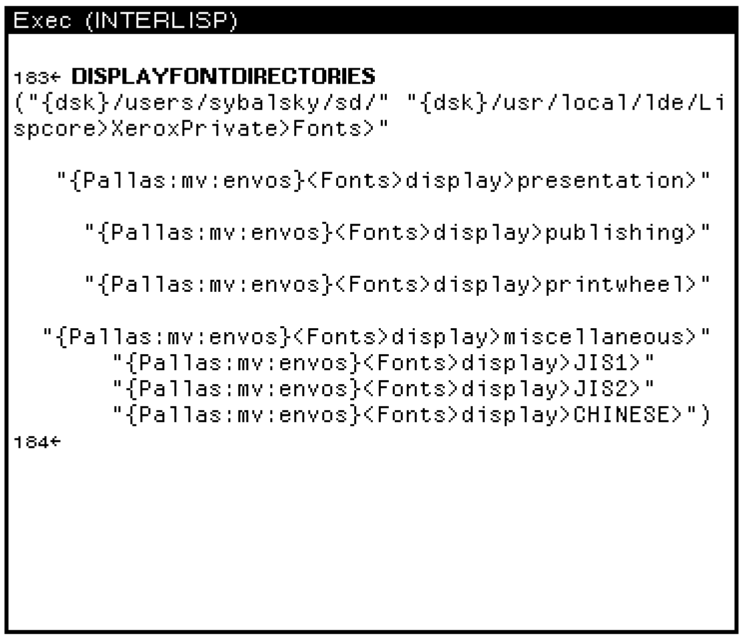


Figure 16-1. Value for the Atom DISFLAYFONTDIRECTORIES

InterPress Fonts

InterPress is the format that is used by Xerox laser printers. These printers normally have a resolution that is much higher than that of the screen: 300 points per inch.

To format files appropriately for output on such a printer, Interlisp must know the actual size for each character that is to be printed. This is done through the use of width files that contain font width information for fonts in InterPress format. For InterPress fonts, you should make the location of these files one of the values of the variable INTERPRESSFONTDIRECTORIES. Its value is a list of directories to search for the font widths files for InterPress fonts. Figure 16-2 is an example value of INTERPRESSFONTDIRECTORIES:

```
Exec (INTERLISP)

184← INTERPRESSFONTDIRECTORIES
("{dsk}/users/sybalsky/sd/" "{dsk}/usr/local/lde/Li
spcore>XeroxPrivate>Fonts>"
       "{Pallas:mv:envos}<Fonts>interpress>presenta
tion>"

  "{Pallas:mv:envos}<Fonts>interpress>publishing>"

  "{Pallas:mv:envos}<Fonts>interpress>printwheel>"
       "{Pallas:mv:envos}<Fonts>interpress>miscella
neous>"
       "{Pallas:mv:envos}<Fonts>interpress>JIS1>"
       "{Pallas:mv:envos}<Fonts>interpress>JIS2>"

     "{Pallas:mv:envos}<Fonts>interpress>CHINESE>"
       )
185←
```

Figure 16-2. Value for Atom INTERPRESSFONTDIRECTORIES

Functions for Using Fonts

FONTPR0P Looking at Font Properties

It is possible to see the properties of a fontdescriptor. This s done with the function FONTPROP. For the following examples, the fontdescriptor used will be the one returned by the function (DEFAULTFONT 'DISPLAY). In other words, the fontdescriptor examined will be the default display font for the system.

There are many properties of a font that might be useful for you. Some of these are:

FAMILY — To see the family of a font descriptor, type:

(FONTPROP (DEFAULTFONT 'DISPLAY) 'FAMILY)

SIZE — As above, this is a positive integer that determines the height of the font in printer's points. As an example, the SIZE of the current default font is:

```
Exec (INTERLISP)

129← (FONTPROP (DEFAULTFONT 'DISPLAY) 'SIZE)
10
130←
```

Figure 16-3. Value of Font Property SIZE of Default Font

ASCENT — The value of this property is a positive integer, the maximum height of any character in the specified font from the baseline (bottom). The top of the tallest character in the font, then, will be at (BASELINE + ASCENT - 1). For example, the ASCENT of the default font is:

Figure 16-4. Value Font Property ASCENT of Default Font

DESCENT — The DESCENT is an integer that specifies the maximum number of points that a character in the font descends below the baseline (e.g., letters such as "p" and "g" have tails that descend below the baseline.). The bottom of the lowest character in the font will be at (BASELINE - DESCENT). To see the DESCENT of the default font, type:

```
(FONTPROP (DEFAULTFONT 'DISPLAY) 'DESCENT)
```

HEIGHT — HEIGHT is equal to (DESCENT - ASCENT).

FACE — The value of this property is a list of the form (*weight slope expansion*). These are the weight, slope, and expansion described above. You can see each one separately, also. Use the property that you are interested in, WEIGHT, SLOPE, or EXPANSION, instead of FACE as the second argument to FONTPROP.

For other font properties, see Chapter 27 of the *IRM*.

STRINGWIDTH

It is often useful to see how much space is required to print an expression in a particular font. The function STRINGWIDTH does this. For example, type:

```
(STRINGWIDTH "Hi there!" (FONTCREATE 'GACHA 10 'STANDARD))
```

The number returned is how many left to right pixels would be needed if the string were printed in this font. (Note that this doesn't just work for pixels on the screen, but for all kinds of streams. For more information about streams, see Chapter 15.) Compare the number returned from the example call with the number returned when you change GACHA to TIMESROMAN.

DSPFONT - Changing the Font in One Window

The function DSPFONT changes the font in a single window. As an example of its use, first create a window to write in. Type:

```
(SETQ MY.FONT.WINDOW (CREATEW))
```

in the Executive Window. Sweep out the window. To print something in the default font, type:

```
(PRINT 'HELLO MY.FONT.WINDOW)
```

in the Executive Window. Your window, MY.FONT.WINDOW, will look something like Figure 16-5:

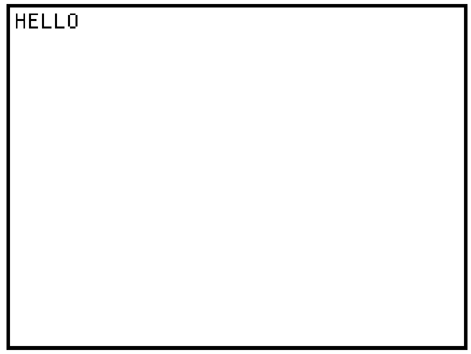
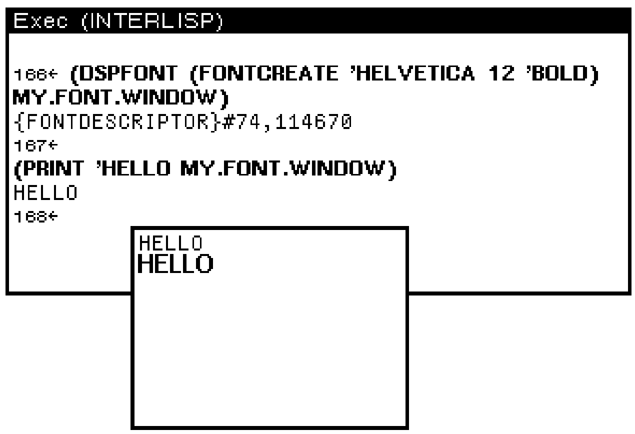


Figure 16-5. HELLO, Printed with the Default Font in MY.FONT.WINDOW

Now change the font in the window. Type:

```
(DSPFONT (FONTCREATE 'HELVETICA 12 'BOLD) MY.FONT.WINDOW)
```

in the Executive Window. The arguments to FONTCREATE can be changed to create any desired font. Now retype the PRINT statement, and your window will look something like Figure 16-6:



Flgure 16-6. Font in MY.FONT.WINDOW Changed

Notice the font has been changed.

Personalizing Your Font Profile

Medley keeps a list of default font specifications. This list is used to set the font in all windows where the font is not specifically set by the user (see the DSPFONT section above). The value of the atom FONTPROFILE is this list (see Figure 16-7).

A FONTPROFILE is a list of font descriptions that certain system functions access when printing output. It contains specifications for big fonts (used when pretty printing a function to type the function name), small fonts (used for printing comments in the editor), and various other fonts.

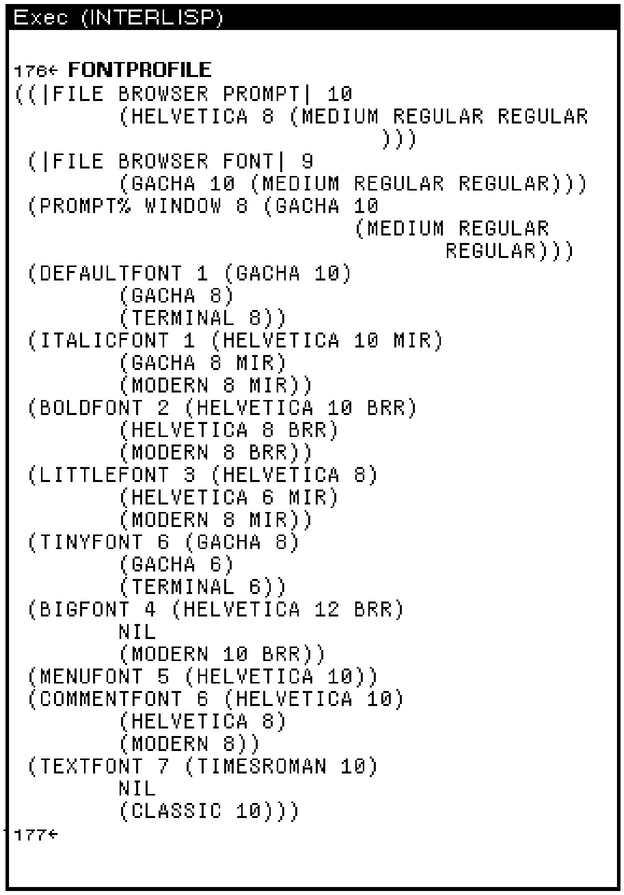


Figure 16-7. Value of the Atom FONTPROFILE

The list is in the form of an association list. The font class names (e.g., DEFAULTFONT, or BOLDFONT) are the keywords of the association list. When a number follows the keyword, it is the font number for that font class.

The lists following the font class name or number are the font specifications, in a form that the function FONTCREATE can use. The first font specification list affer a keyword is the specification for printing to windows. The list (GACHA 10) in the figure above is an example of the default specification for the printing to windows. The last two font specification lists are for Press and InterPress file printing, respectively. For more information, see Chapter 27 in the *IRM*.

Now, to change your default font settings, change the value of the variable FONTPROFILE. Medley has a list of profiles stored as the value of the atom FONTDEFS. Choose the profile to use, then install it as the default FONTPROFILE.

Evaluate the atom FONTDEFS and notice that each profile list begins with a keyword (see Figure 16-8). This keyword corresponds to the size of the fonts included. BIG, SMALL, and STANDARD are some of the keywords for profiles on this list—SMALL and STANDARD appear in Figure 16-8.

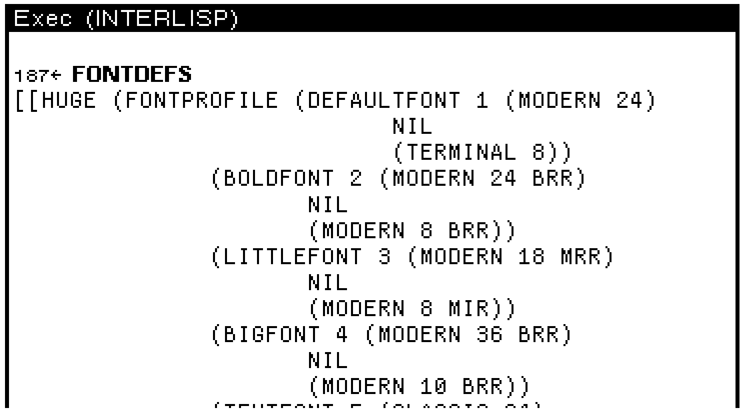


Figure 16-8. Part of Value of the Atom FONTDEFS

To install a new profile from this list, follow the following example, but insert any keyword for BIG.

To use the profile with the keyword BIG instead of the standard one, evaluate the following expression:

```
(FONTSET 'BIG))
```

Now the fonts are permanently replaced. (That is, until another profile is installed.)

17. THE INSPECTOR

The Inspector is a window-oriented tool designed to examine data structures. Because Medley is such a powerful programming environment, many types of data structures would be difficult to see in any other way.

Calling the Inspector

Take as an example an object defined through a sequence of pointers (i.e., a bitmap on the property list of a window on the property list of an atom inaprogram.)

To inspect an object named `NAME`, type:

```
(INSPECT 'NAME)
```

If `NAME` has many possible interpretations, an option menu will appear. For example, in Interlisp-D, a litatom can refer to both an atom and a function. For example, if `NAME` was a record, had a function definition, and had properties on its property list, then the menu would appear as in Figure 17-1.

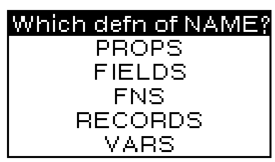


Figure 17-1. Option Window for Inspection of `NAME`

If `NAME` were a list, then the option menu shown in Figure 17.2 would appear. The options include:

- Calling the display editor on the list
- Calling the TTY editor (see Chapter 6)
- Seeing the list's elements in a display window. If you choose this option, each element in the list will appear in the right column of the Inspector window. The left column of the Inspector window will be made up of numbers (see Figure 17-3).
- Inspecting the list as a record type (this last option would produce a menu of known record types). If you choose a record type, the items in the list will appear in the right column of the Inspector window. The left column of the Inspector window will be made up of the field names of the record.

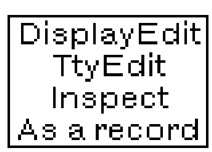


Figure 17-2. Option Window for Inspection of List

Using the Inspector

If you choose to display your data structure in an edit window, simply edit the structure and exit in the normal manner when done. If you choose to display the data structure in an inspect window, then follow these instructions:

- To select an item, point the mouse cursor at it and press the left mouse button.
- Items in the right column of an Inspector window can themselves be inspected. To do this, choose the item, and press the center mouse button.
- Items in the right column of an Inspector window can be changed. To do this, choose the corresponding item in the left column, and press the center mouse button. You will be prompted for the new value, and the item will be changed. The sequence of steps is shown in Figure 17-3.

The item in the lefl column is selected, and the middle mouse button pressed. Select the `SET` option from the menu that pops up.

You will then be prompted for the new value. Type it in.

The item in the right column is updated to the value of what you typed in.

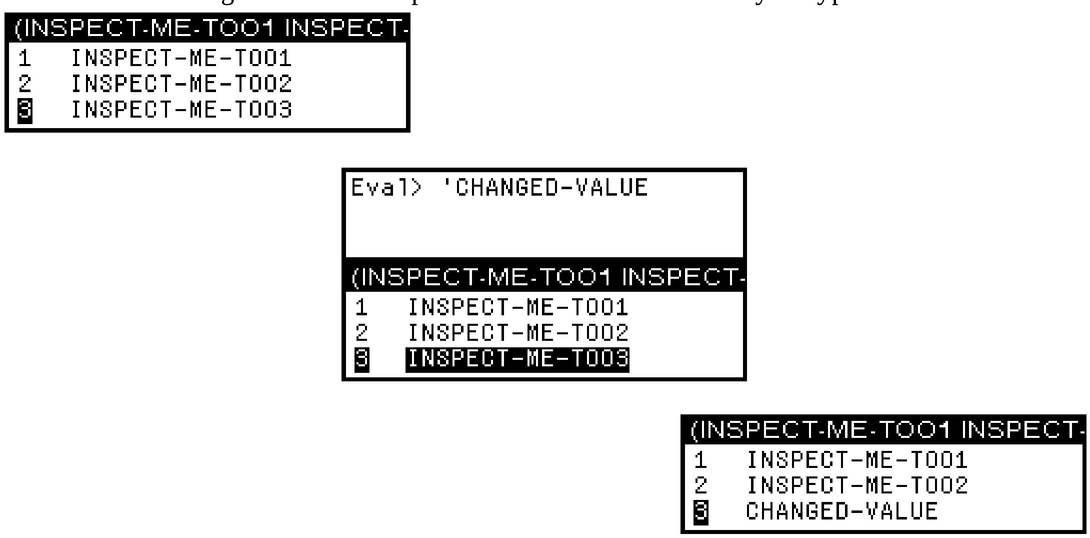


Figure 17-3. Steps Involved in Changing Value in Right Column of Inspector Window

Inspector Example

This example will use ideas discussed in Chapter 21. An example, `ANlMALGRAPH`, is created in that section. You do not need to know the details of how it was created, but the structure is examined in this chapter.

If you type

```
(INSPECT ANIMAL.GRAPH)
```

and then choose the Inspect option from the menu, a display appears as shown in Figure 17-4. `ANlMAL.GRAPH` is being inspected as a list. Note the numbers in the left column of the inspectorwindow.

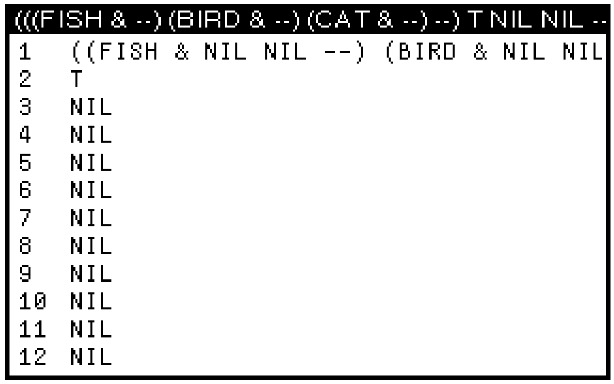


Figure 17-4. Inspector Window For ANIMAL.GRAPH, Inspected as List

If you choose the "As A Record" option, and choose "GRAPH" from the menu that appears, the inspector window looks like Figure 17-5. Note the fieldnames in the left column of the inspector window.

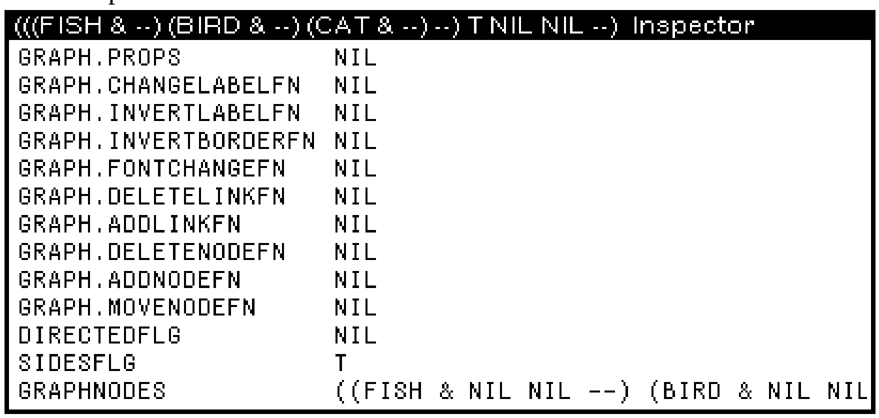


Figure 17-5. Inspector Window for ANlMAL.GRAPH, Inspected as Instance of GRAPH Record

The remaining examples will use ANlMAL.GRAPH inspected as a list. When the first item in the Inspector window is chosen with the leff mouse button, the Inspector window looks like Figure 17-6.

Figure 17-6. Inspector Window for ANlMAL.GRAPH With First Element Selected

When you use the middle mouse button to inspect the selected list element, the display looks like Figure 17-7.

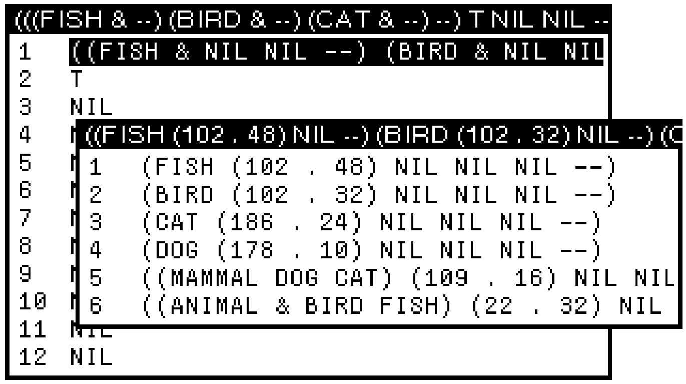


Figure 17-7. Inspector Window for `AN1MAL.GRAPH` and for First Element of `ANIMAL.GRAPH`

How you can see that six items make up the list, and you can further choose to inspect one of these items. Notice that this is also inspected as a list. As usual, it could also have been inspected as a record.

Select item 5 - `MAMMAL DOG CAT` - with the left mouse button. Press the middle mouse button. Choose "Inspect" to inspect your choice as a list. The Inspector now displays the values of the structure that makes up `MAMMAL DOG CAT`. (See Figure 17-8.)

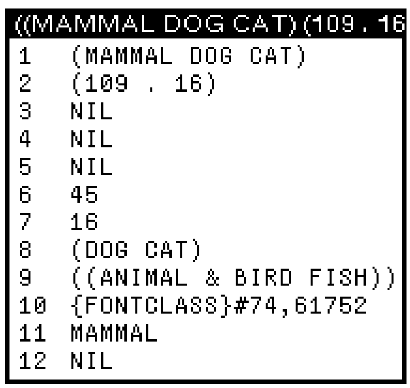


Figure 17-8. Inspector Window for Element S From Figure 17.7 That Begins `((MAMMAL DOG CAT)`.

18. MASTERSCOPE

Masterscope is a tool that allows you to quickly examine the structure of complex programs. As your programs enlarge, you may forget what variables are global, what functions call other functions, and so forth. Masterscope keeps track of this for you.

To use Masterscope, first load `MASTERSCOPE.DFASL` and `EXPORTS.ALL`.

Suppose that `JVTO` is the name of a file that contains many of the functions involved in a complex system and that `LINTRANS` is the file containing the remaining functions. The first step is to ask Masterscope to analyze these files. These files must be loaded. All Masterscope queries and commands begin with a period followed by a space, as in

```
. ANALYZE FNS ON MSCOPEDEMO
```

The `ANALYZE` process takes a while, so the system prints a period on the screen for each function it has analyzed. (See Figure 18-1)

Figure 18-1. Executive Window After Analyzing Files

If you are not quite sure what functions were just analyzed, type the file's `COMS` variable (see the File Variables section in Chapter 7) into the Executive Window. The names of the functions stored on the file will be a part of the value of this variable.

A variety of commands are now possible, all referring to individual functions within the analyzed files. Substantial variation in exact wording is permitted. Some commands are:

`. SHOW PATHS FROM ANY TO ANY`
`. EDIT WHERE ANY CALLS` *functionname*
`. EDIT WHERE ANY USES` *variablename*
`. WHO CALLS WHOM`
`. WHO CALLS` *functionname*
`. BY WHOM IS` *functionname* `CALLED`
`. WHO USES` *variablename* `AS FIELD`

Note that the function is being called to invoke each command. Refer to the *IRM* for commands not listed here.

Figure 18-2 shows the Executive Window after the commands `. WHO CALLS GobbleDump` and `. WHO DOES JVL inScan CALL`.

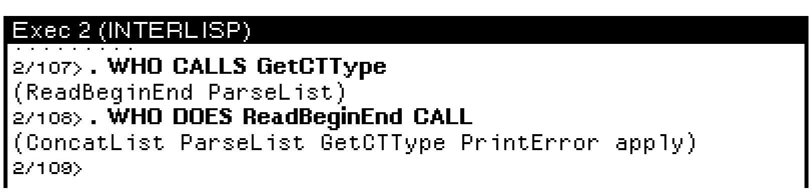


Figure 18-2. Sample Masterscope Output

SHOW DATA Command and GRAPHER

When the library package GRAPHER is loaded (to load this package, type (FILESLOAD GRAPHER)), Masterscope's SHOWPATHS command is modified. The command will be changed to generate a tree structure showi ng how the program's functions interact instead of a tabular printout into the Executive window. For example, typing:

```
. SHOW PATHS FROM ProcessEND
```

produced the display shown in Figure 18-3.

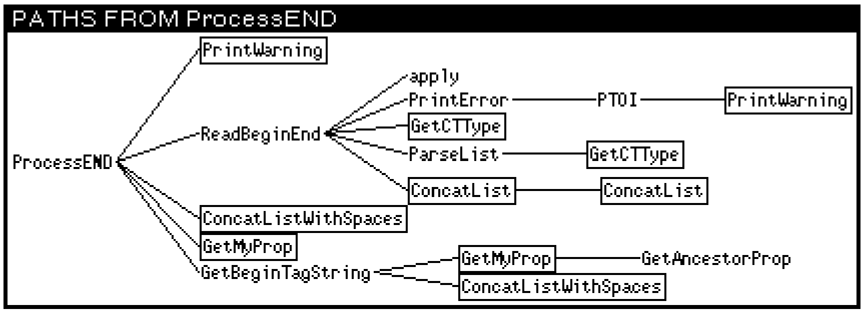


Figure 18-3. SHOW PATHS Display Example

All the functions in the display are part of this analyzed file or a previously analyzed file. Boxed functions indicate that the function name has been duplicated in another place on the display.

Selecting any function name on the display will pretty print the function in a window (see Figure 18-4).

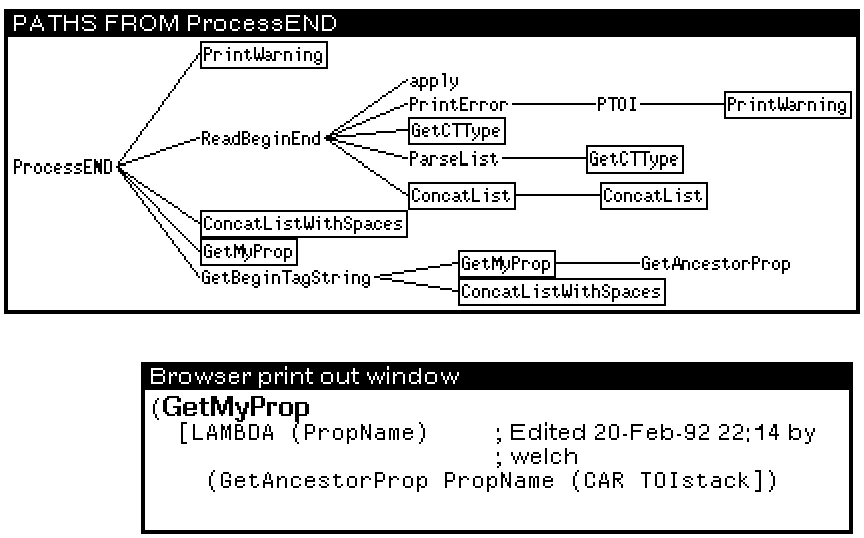


Figure 18-4. Browser Printout Example

Selecting it again with the left mouse button will produce a desription of the function's role in the overall system (see Figure 18-5).

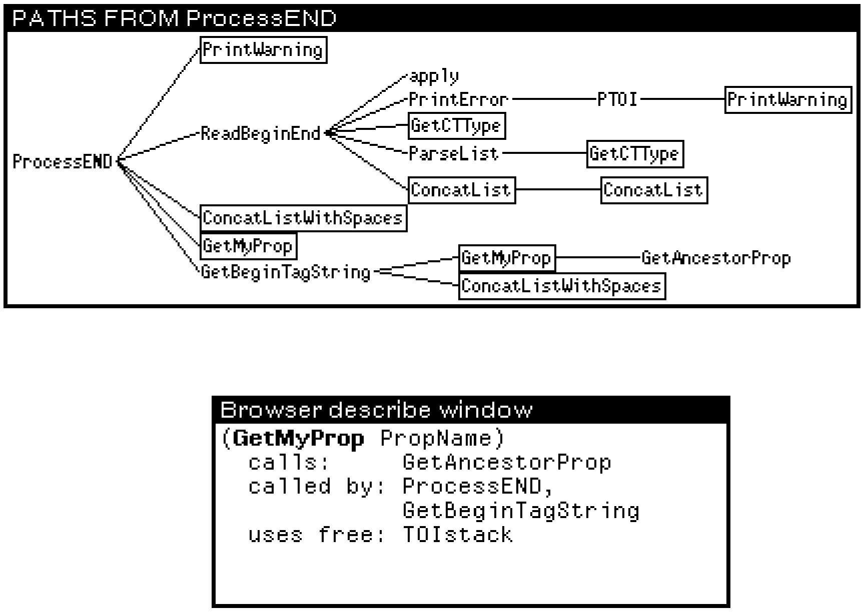


Figure 18-5. Browser Description Example

19. WHERE DOES ALL THE TIME GO? SPY

SPY is an Lisp library package that shows you where you spend your time when you run your system. It is easy to learn, and very useful when trying to make programs run faster.

How to Use Spy with the SPY Window

The function SPY.BUTTON brings up a small window which you will be prompted to position. Using the mouse buttons in this window controls the action of the SPY program. When you are not using SPY, the window appears as in Figure 19.1.

Figure 19.1. SPY Window When SPY is Not Being Used

To use SPY, click either the left or middle mouse button with the mouse cursor in the SPY window. The window will appear as in Figure 19.2, and means that SPY is accumulating data about your program.

Figure 19.2. SPY Window When SPY is Being Used

To turn off SPY after the program has run, again click a mouse button in the SPY window. The eye closes, and you are asked to position another window. This window contains SPY's results. An example of the resulting window is shown in Figure 19.3.

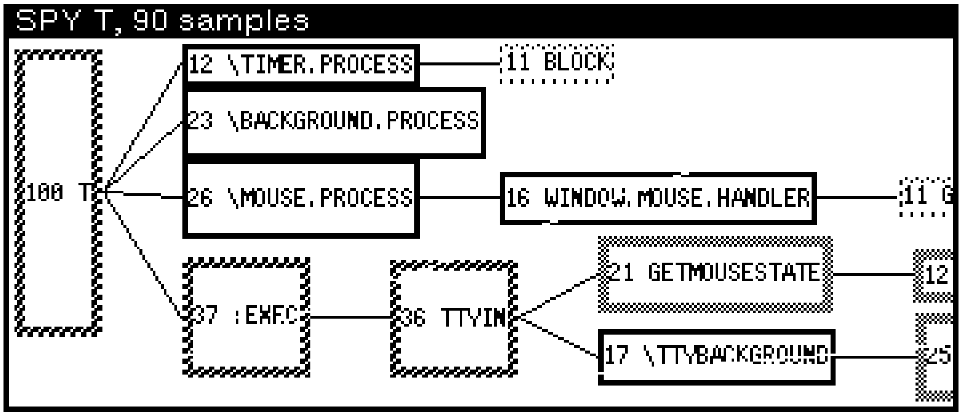


Figure 19.3. Window Produced After Running SPY

This window is scrollable horizontally and vertically. This is useful, since the whole tree does not fit in the window. If a part that you want to see is not shown, you can scroll the window to show the part you want to see.

How to Use SPY from the Lisp Top Level

SPY can also be run while a specific function or system is being used. To do this, type the function WITH.SPY:

(WITH.SPY *form*)

The expression used for form should be the call to begin running the function or system that SPY is to watch. If you watch the SPY window, the eye will blink! To see your results, run the function SPY.TREE. To do this, type:

(SPY.TREE)

The results of the last running of SPY will be displayed. If you do this, and SPY.TREE returns (no SPY samples have been gathered), your function ran too fast for SPY to follow.

Interpreting SPY's Results

Each node in the tree is a box that contains, first, the percentage of time spent running that particular function, and second, the function name. There are two modes that can be used to display this tree.

The default mode is cumulative. In this mode, each percentage is the amount of time that function spent on top of the stack, plus the amount of time spent by the functions it calls. The second mode is individual. To change the mode to individual, point to the title bar of the window, and press the middle mouse button. Choose Individual from the menu that appears. In this mode, the percentage shown is the amount of time the function spent on the top of the stack.

To look at a single branch of the tree, point with the mouse cursor at one of the nodes of the tree, and press the right mouse button. From the menu that appears, choose the option `SubTree`. Another `SPY` window will appear, with just this branch of the tree in it.

Another way to focus within the tree is to remove branches from the tree. To do this, point to the node at the top of the branch you would like to delete. Press the middle mouse button, and choose `Delete` from the menu that appears.

There are also different amounts of "merging" of functions that can be done in the window. A function can be called by another function more than once. The amount of merging determines where the subfunction, and the functions that it calls, appear in the tree, and how often. (For a detailed explanation of merging, see the *Lisp Library Packages Manual*.)

20. FREE MENUS

Free Menu is a library package that is even more flexible than the regular menu package. It allows you to create menus with different types of items in them, and formats them as you require. Free menus are particularly useful when you want a "fill in the form" type interaction with the user.

Each menu item is described with a list of properties and values. The following example will give you an idea of the structure of the description list, and some of your options. The most commonly used properties, and each type of menu item will be described in the Parts of a Free Menu Item and Types of Free Menu Items section below.

Free Menu Example

Free menus can be created and formatted automatically! It is done with the function FM.FORMATMENU. This function takes one argument, a description of the menu. The description is a list of lists; each internal list describes one row of the free menu. A free menu row can have more than one item in it, so there are really lists of lists of lists! It really isn’t hard, though, as you can see from the following example:

```
(SETQ ExampleMenu
    (FM.FORMATMENU
        ’(((TYPE TITLE LABEL TitlesDoNothing)
            TYPE 3STATE LABEL Example3State))
          ((TYPE EDITSTART LABEL PressToStartEditing
                           ITEMS (EDITEM))
           (TYPE EDIT ID EDITEM LABEL ""))
        (WINDOWPROPS TITLE "Example Does Nothing"))))
```

The first row has two items in it: one is a TITLE, and the second is a 3STATE item. The second row also has two items. The second, the EDIT item, is invisible, because its label is an empty string. The caret will appear for editing, however, if the EDITSTART item is chosen. Windowprops can appear as part of the description of the menu, because a menu is, affer all, just a special window. You can specify not only the title with WINDOWPROPS, but also the position of the free menu, using the "left" and "bottom" properties, and the width of the border in pixels, with the "border" property. Evaluating this expression will return a window. You can see the menu by using the function OPENW. The following example illustrates this:

Figure 20.1. Example Free Menu

The next example shows you what the menu looks like after the EDITSTART item, PressToStartEditing, has been chosen.

Figure 20.2. Free menu after EDITSTART Item Chosen

The following example shows the menu with the 3STATE item in its T state, with the item highlighted. (In the previous bitmaps, it was in its neutral state.)

.

Figure 20.3. Free menu with 3STATE Item in its T State

Finally, Figure 20.4 shows the 3STATE item in its NIL state, with a diagonal line through the item

Figure 20.4 Free menu with the 3STATE item in its NIL State

If you would like to specify the layout yourself, you can do that too. See the *Lisp Library Packages Manual* for more information.

Parts of a Free Menu Item

There are eight different types of items that you can use in a free menu. No matter what type, the menu item is easily described by a list of properties, and values. Some of the properties you will use most often are listed below:

LABEL — Required for every type of menu item. It is the atom, string, or bitmap that appears as a menu selection.

TYPE — One of eight types of menu items. Each of these are described in the section below.

MESSAGE — The message that appears in the prompt window if a mouse button is held down over the item.

ID — An item's unique identifier. An ID is needed for certain types of menu items.

ITEMS — Used to list a series of choices for an NCHOOSE item, and to list the ID's of the editable items for an EDITSTART item.

SELECTEDFN — The name of the function to be called if the item is chosen.

Types of Free Menu Items

Each type of menu item is described in the following list, including an example description list for each one.

MOMENTARY — This is the familiar sort of menu item. When it is selected, the function stored with it is called. A description for the function that creates and formats the menu looks like this:

```
(TYPE MOMENTARY
    LABEL Blink-N-Ring
    MESSAGE "Blinks the screen and rings bells"
    SELECTEDFN RINGBELLS)
```

TOGGLE — This menu item has two states, T and NIL. The default state is NIL, but choosing the item toggles its state. The following is an example description list, without code for the SELECTEDFN function, for this type of item:

```
(TYPE TOGGLE
    LABEL DwimDisable
    SELECTEDFN ChangeDwimState)
```

3STATE

This type of menu item has three states, NEUTRAL, T, and NIL. NEUTRAL is the default state. T is shown by highlighting the item, and NIL is shown with diagonal lines. The following is an example description list, without code for the SELECTEDFN function, for this type of item:

```
(TYPE 3STATE
    LABEL CorrectProgramAllOrNoSpelling
    SELECTEDFN ToggleSpellingCorrection)
```

TITLE

This menu item appears on the menu as dummy text. It does nothing when chosen. An example of its description:

```
(TYPE TITLE LABEL "Choices:")
```

NWAY

A group of items, nnly one of which can be chosen at a time. The items in the NWAY group should all have an ID field, and the ID's should be the same. For example, to set up a menu that would allow the user to choose between Helvetica, Gacha, Modern, and Classic fonts, the descriptions might look like this (once again, without the code for the SELECTEDFN):

```
(TYPE NWAY ID FONTCHOICE
    LABEL Helvetica
    SELECTEDFN ChangeFont)
(TYPE NWAY ID FONTCHOICE
    LABEL Gacha
    SELECTEDFN ChangeFont)
(TYPE NWAY ID FONTCHOICE)
    LABEL Modern
    SELECTEDFN ChangeFont)
(TYPE NWAY ID FONTCHOICE
    LABEL Classic
    SELECTEDFN Changefont)
```

NCHOOSE

This type of menu item is like NWAY except that the choices are given to the user in a submenu. The list to specify an NCHOOSE menu item that is analogous to the NWAY item above might look like this:

```
(TYPE NCHOOSE
    LABEL FontChoices
    ITEMS Helvetica Gacha Modern Classic)
    SELECTDFN Changefont)
```

EDITSTART

When this type of menu itein is chosen, it activates another type of item, an EDIT item. The EDIT item or items associated with an EDITSTART item have their lD's listed on the EDITSTART's ITEMS property. An example description list is:

```
(TYPE EDITSTART LABEL "Function to add?" ITEMS (Fn))
```

EDIT

This type of menu item can actually be edited by you. It is often associated with an EDITSTART item (see above), but the caret that prompts for input will also appear if the item itself is chosen. An EDIT item follows the same editing conventions as editing in Executive Window:

Add characters by typing them at the caret.

Move the caret by pointing the mouse at the new position, and clicking the left button.

Delete characters from the caret to the mouse by pressing the right button of the mouse. Delete a character behind the caret by pressing the backspace key.

Stop editing by typing a carriage return, a Control-X, or by choosing another item from the menu.

An example description list for this type of item is:

```
(TYPE EDIT ID Fn LABEL **)
```

21. THE GRAPHER

Say it with Graphs

Grapher is a collection of functions for creating and displaying graphs, networks of nodes and links. Grapher also allows you to associate program behavior with mouse selection of graph nodes. To load this package, type

```
(FILESLOAD GRAPHER)
```

Figure 21-1 shows a simple graph.

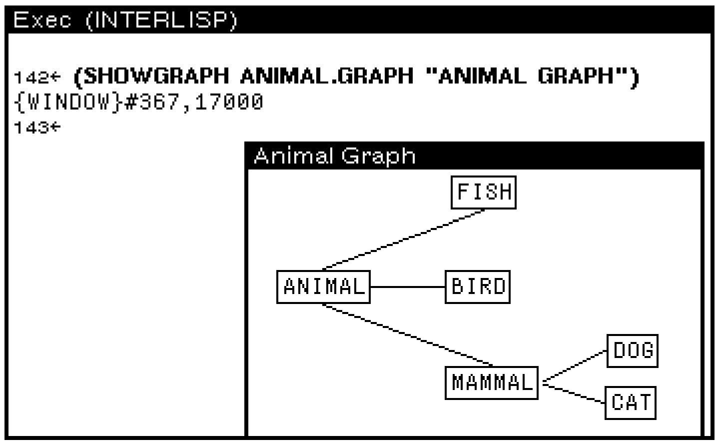


Figure 21-1. Simple Graph

In Figure 21-1 there are six nodes (ANIMAL, MAMMAL, DOG, CAT, FISH, and BIRD) connected by five links. A GRAPH is a record containing several fields. Perhaps the most important field is GRAPHNODES—which is itself a list of GRAPHNODE records. Figure 21-2 illustrates these data structures. The window on top contains the fields from the simple graph. The window on the bottoms an inspection of the node, DOG.

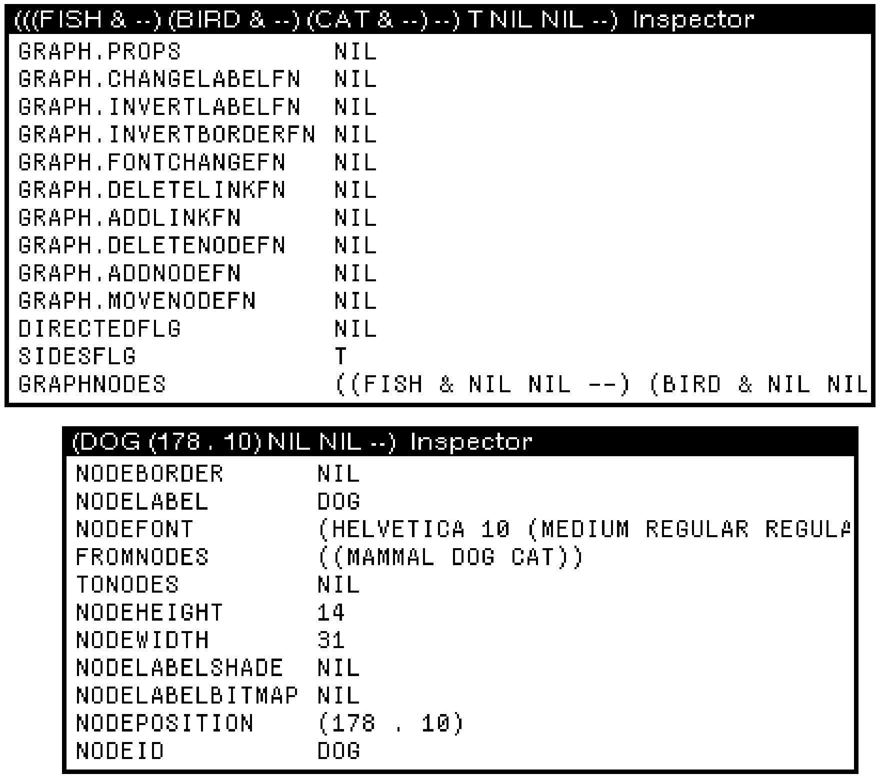


Figure 21-2. Inspecting a Graph and a Node

The GRAPHNODE data structure is described by its text (NODEID), what goes into it (FROMNODES), what leaves it (TONODES), and other fields that specify its looks. The basic model of graph building is to create a bunch of nodes, then layout the nodes into a graph, and finally display the resultant graph. This can be done in a number of ways. One is to use the function NODECREATE to create the nodes, LAYOUTGRAPH to lay out the nodes, and SHOWGRAPH to display the graph. The primer shows you two simpler ways, but please see the *Library Packages Manual* for more information about these other functions. The primer's first method is to use SHOWGRAPH to display a graph with no nodes or links, then interactively add them. The second is to use the function LAYOUTSEXPR, which does the appropriate NODECREATES and a LAYOUTGRAPH, with a list.

The function SHOWGRAPH displays graphs and allows you to edit them. The syntax of SHOWGRAPH is

(SHOWGRAPH *graph window lefibuttonfn middlebuttonfn*
topjustifyflg alloweditflg copybuttoneventfn)

Obviously the graph structure is very complex. Here's the easiest way to create a graph.

```
(SETQ MY.GRAPH NIL)
(SHOWGRAPH MY.GRAPH "My Graph" NIL NIL NIL T)
```

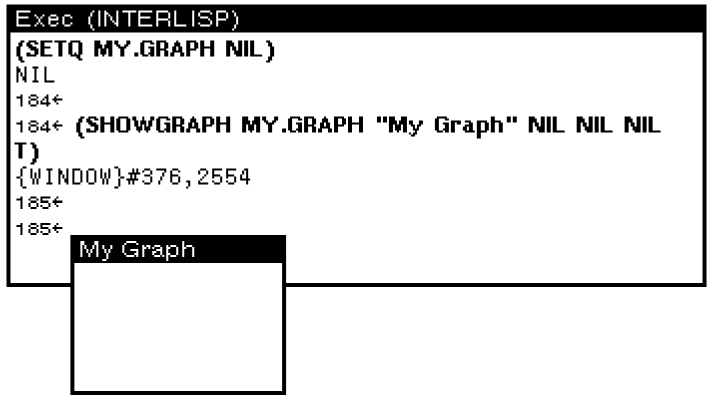


Figure 21-3. My Graph

You will be prompted to create a small window as in Figure 21-3. This graph has the title My Graph. Hold down the right mouse button in the window. A menu of graph editing operations will appear as in Figure 21-4.

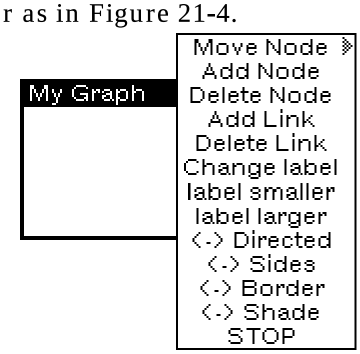


Figure 21-4. Menu of Graph Editing Operations

Here's how to use this menu. The commands in this menu are easy to learn. Experiment with them!

Add a Node

Start by selecting `Add Node`. Grapher will prompt you for the name of the node (see Figure 21-5.) and then its position.

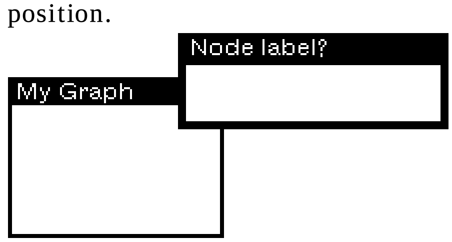


Figure 21-5. Grapher Prompts for Name of Node to add after Add Node is Chosen from Graph Editing Menu.

Position the node by moving the mouse cursor to the desired location and clicking a mouse button. Figure 21-6 shows the graph with two nodes added using this menu.

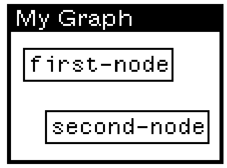


Figure 21-6. Two Nodes Added to `MY GRAPH` Using GraphEditing Menu

Add a Link

Select `Add Link` from the graph editing menu. The Prompt window will prompt you to select the two nodes to be linked. (See Figure 21-7.) Do this, and the link will be added.

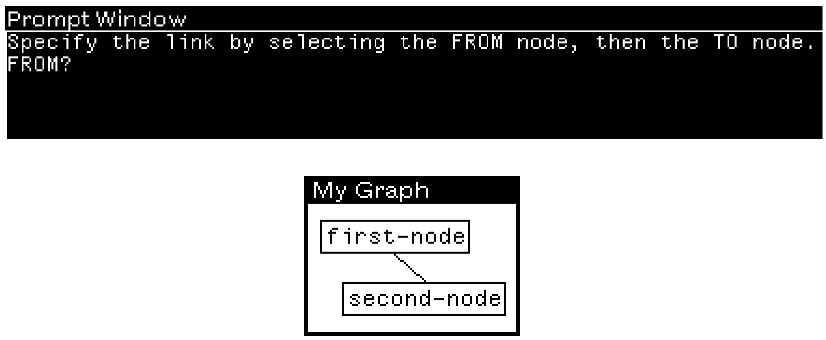


Figure 21-7. Prompt Window Requesting Selection of Two Nodes to Link, and Result

Delete a Link

Select `Delete Link` from the graph editing menu. ThePrompt window will prompt you to select the two nodes that should no longer be linked. (See Figure 21-8.) Do this, and the link will be deleted.

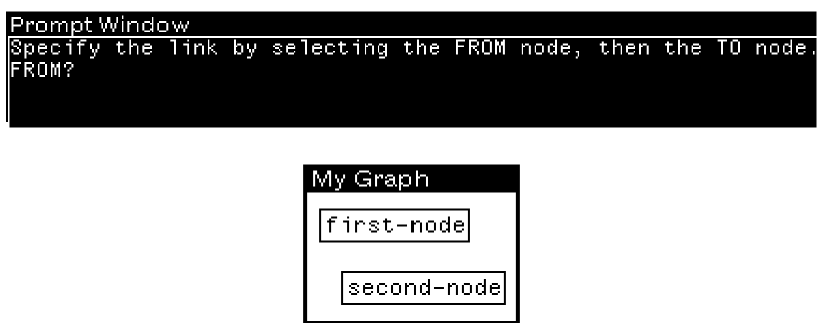


Figure 21-8. Prompt Window Requesting Selection of Link to Delete, and Result

Delete a Node

Select `Delete Node` from the graph editing menu. The Prompt window will prompt you to select the node to be aeleted. (See Figure 21-9.) Do this, and the node will be deletea.

Figure 21.-9. Prompt to Delete a Node

Move a Node

Select `Delete Node` from the graph editng menu. Choose a node pointing to the it with the mouse cursor, and pressing and holding the leff mouse button. When you move the mouse cursor, the node will be dragged along. When the node is at the new position, release the mouse button to deposit the node.

Making a Graph from a List

Typically, a graph is used to display one of your program's data structures. Here is how that is done.

LAYOUTSEXPR takes a list and returns a GRAPH record. The syntax of the function is

(LAYOUTSEXPR *sexpr format boxing font motherd personald famlyd*)

For example:

```
(SETQ ANIMAL.TREE '(ANIMAL (MAMMAL DOG CAT) BIRD FISH))
(SETQ ANIMAL.GRAPH
     (LAYOUTSEXPR ANIMAL.TREE 'HORIZONTAL))
(SHOWGRAPH ANIMAL.GRAPH "My Graph" NIL NIL NIL T)
```

This is how Figure 21.1 was produced.

Incorporating Grapher into Your Program

The Grapher is designed to be built into other programs. It can call functions when, for example, a mouse button is clicked on a node. The function SHOWGRAPH does this:

(SHOWGRAPH *graph window lefibuttonfn middlebuttonfn*
topjustifyflg alloweditflg copybuttoneventfn)

For example, the third argument to SHOWGRAPH, *leftbuttonfn*, is a function that is called when the left mouse button is pressed in the graph window. Try this:

```
(DEFINEQ (My.LEFT.BUTT0N.FUNCTION
     (THE.GRAPHNODE THE.GRAPH.WINDOW)
          (INSPECT THE.GRAPHNODE)))


(SHOWGRAPH FAMILY.GRAPH "Inspectable family"
     (FUNCTION MY.LEFT.BUTTON.FUNCTION)
          NIL NIL T)
```

In the example above, MY.LEFT.BUTTON.FUNCTION simply calls the inspector. The function should be written assuming it will be passed a graphnode and the window that holds the graph. Try adding a function of your own.

More of Grapher

Some other Library packages make use of the Grapher. (Grapher needs to be loaded with the packages to use these functions.)

- `MASTERSCOPE`: The Browser package modifies the Masterscope command, . `SHOW PATHS`, so that its output is displayed as a graph (using Grapher) instead of simply printed.
- `GRAPHZOOM`: allows a graph to be redisplayed larger or smaller automatically.

22. RESOURCE MANAGEMENT

Naming Variables and Records

You will find times when one environment simultaneously hosts a number of different programs. Running a demo of several programs, or reloading the entire Medley environment from floppies when it contains several different programs, are two examples that could, if you aren't careful, provide a few problems. Here are a few tips on how to prevent problems:

- If you change the value of a system variable, MENUHELDWAIT for example, or connect to a directory other than {DSK}<LISPFILES>, write a function to reset the variable or directory to its original value. Run this function when you are finished working. This is especially important if you change any of the system menus.

- Do not redefine Medley functions or CLISP words. Remember, if you reset an atom's value or function definition at the top level (in the Executive Window), the message (*Some.Crucial.Function.Or.Variable* redefined), appears. If this is not what you wanted, type **UNDO** immediately!

  If, however, you reset the value or function definition of an atom inside your program, a warning message will not be printed.

- Make the atom names in your programs as unique as possible. To do this without filling your program with unreadable names that noone, including you, can remember, prefix your variable names with the initials of your program. Even then, check to see that they are not already being used with the function BOUNDP. For example, type:

  ```
  (BOUNDP 'BackgroundMenu)
  ```

  This atom is bound to the menu that appears when you press the leff mouse button when the mouse cursor is not in any window. BOUNDP returns T. BOUNDP returns NIL if its argument does not currently have a value.

- Make your function names as unique as possible. Once again, prefixing function names with the initials of your program can be helpful in making them unique, but even so, check to see that they are not already being used. GETD is the Interlisp-D function that returns the function definition of an atom, if it has one. If an atom has no function definition, GETD returns NIL. For example, type:

  ```
  (GETD 'CAR)
  ```

  A non-NIL value is returned. The atom CAR already has a function definition.

- Use complete record field names in record FETCHes and REPLACEs when your code is not compiled. A complete record field name is a list consisting of the record declaration name and the field name. Consider the following example:

  ```
  (REC0RD NAME (FIRST LAST))
  (SETQ MyName (create Name FIRST←'John LAST←'Smith))
  (FETCH (NAME FIRST) OF MyName)
  ```

- Avoid reusing names that are field names of Lisp system records. A few examples of system records follow. Do not reuse these names.

  ```
  (RECORD REGION (LEFT BOTTOM WIDTH HEIGHT))
  (RECORD POSITION (XCOORD YCOORD))
  ```

```
(RECORD IMAGEOBJ (- BITMAP -)))
```

- When you select a record name and field names for a new record, check to see whether those names have already been used.

  Call the function RECLOOK, with your record name as an argument, in the Executive Window (see Figure 22-1). If your record name is already a record, the record definition will be returned; otherwise the function will return NIL.

```
Exec 2 (INTERLISP)
NIL
2/170> (RECLOOK 'POSITION)
(RECORD POSITION (XCOORD . YCOORD)
       [TYPE? (AND (LISTP DATUM)
                   (NUMBERP (CAR DATUM))
                   (NUMBERP (CDR DATUM]
       (SYSTEM))
2/171> (RECLOOK 'NewPos)
NIL
2/172>
```

Figure 22-1. Response to RECLOCK

Call the function FIELDLOOK with your new field name in the Executive Window (see Figure 22-2). If your field name is already a field name in another record, the record definition will be returned; otherwise the function will return NIL.

```
Exec 2 (INTERLISP)
NIL
2/172> (FIELDLOOK 'XCOORD)
((RECORD POSITION (XCOORD . YCOORD)
        [TYPE? (AND (LISTP DATUM)
                    (NUMBERP (CAR DATUM))
                    (NUMBERP (CDR DATUM]
        (SYSTEM)))
2/173> (FIELDLOOK 'XPos)
NIL
2/174>
```

Figure 22-2. Response to FIELDLOOK

Some Space and Time Considerations

In order for your program to run at maximum speed, you must efficiently use the space available on the system. The following section points out areas that you may not know are wasting valuable space, and tips on how to prevent this waste.

Often programs are written so that new data structures are created each time the program is run. This is wasteful. Write your programs so that they only create new variables and other data structures conditionally. If a structure has already been created, use it instead of creating a new one.

Some time and space can be saved by changing your RECORD and TYPERECORD declarations to DATATYPE. DATATYPE is used the same way as the functions RECORD and TYPERECORD. In addition, the same FETCH and REPLACE commands can be used with the data structure DATATYPE creates. The difference is that the data structure DATATYPE creates cannot be treated as a list the way RECORDs and TYPERECORDs can.

Global Variables

Once defined, global variables remain until Lisp is reloaded. Avoid using global variables if at all possible! One specific problem arises when programs use the function `GENSYM`. In program development, many atoms are created that may no longer be useful. Hints:

- Use

  `(DELDEF` *atomname* `'PROP)`

  to delete property lists, and

  `(DELDEF` *atomname* `'VARS)`

  to have the atom act like it is not defined.

  These not only remove the definition from memory, but also change the appropriate `fileCOMS` that the deleted object was associated with so that the file package will not attempt to save the object (function, variable, record definition, and so forth) the next time the file is made. Just doing something like

  `(SETQ (`*arg atomname*`) 'NOBIND)`

  looks like it will have the same effect as the second `DELDEF` above, but the `SETQ` does not update the file package.

- If you are generating atom names with `GENSYM`, try to keep a list of the atom names that are no longer needed. Reuse these atom names, before generating new ones. There is a (fairly large) maximum to the number of atoms you can have, but things slow down considerably when you create lots of atoms.
- When possible, use a data structure such as a list or an array, instead of many individual atoms. Such a structure has only one pointer to it. Once this pointer is removed, the whole structure will be garbage-collected and space will be reclaimed.

Circular Lists

If your program is creating circular lists, a lot of space may be wasted. (Many crosslinked data structures end up having circularities.) Hints when using circular lists:

- Write a function to remove pointers that make lists circular when you are through with the circular list.
- If you are working with circular lists of windows, bind your main window to a unique global variable. Write window creation conditionally so that if the binding of that variable is already a window, use it, and only create a new window if that variable is unbound or `NIL`.

Here is an example that illustrates the problem. When several auxilIary windows are built, pointers to these windows are usually kept on the main window's property list. Each auxilIary window also typically keeps a pointer to the main window on its property list If the top level function creates windows rather than reusing existing ones, there will be many lists of useless windows cluttering the work space. Or, if such a main window is closed and will not be used again, you will have to break the links by deleting the relevant properties from both the main window and all of the auxiliary windows first. This is usually done by putting a special `CLOSEFN` on the main window and all of its auxiliary windows.

When You Run Out of Space

Typically, if you generato a lot of structure! that won't get garbage collected, you will eventually run out of space. The important part ii being aNe to track down those structures and the code that generates them to become more space efficient.

Use the Lisp Library Package `GCHAX.DCOM` to track down pointers to data structures. The basic idea is that `GCHAX` will return the number of references to a particular data structure.

A special function exists that allows you to get a little extra space so that you can try to save your work when you get toward the edge (usually noted by a message indicating that you should save your work and load a new Medley environment). The `GAINSPACE` function allows you to delete non-essential data structures. To use it, type:

```
(GAINSPACE)
```

into the Executive Window. Answer `N` to all questions except the followi ng.

- Delete edit history
- Delete history list.
- Delete values of old variables.
- Delete your `MASTERSCOPE` database
- Delete information for undoing your greeting.

Save your work and reload Lisp as soon as possible.

23. SIMPLE INTERACTIONS WITH THE CURSOR, A BITMAP, AND A WINDOW

The purpose of this chapter is to show you how to build a moderately tricky interactive interface with the various Medley display facilities. In particular how to move a large bitmap (larger than 16 x 16 pixels) around inside a window. To do this, you will change the CURSORINFN and CURSOROUTFN properties of the window. If you would also like to then set the bitmap in place in the window, you must reset the BUTTONEVENTFN. This chapter explains how to create the mobile bitmap.

GETMOUSESTATE Example Function

One function that you will use to "trace the cursor" (have a bitmap follow the cursor around in a window) is GETMOUSESTATE. This function finds the current state of the. mouse, and resets global system variables, such as LASTMOUSEX and LASTMOUSEY.

As an example of how this function works, create a window by typing

```
(SETQ EXAMPLE.WINDOW (CREATEW))
```

into the Executive Window, and sweeping out a window. Now, type in the function

```
(DEFINEQ (PRINTCOORDS (W)
     (PROMPTPRINT "(" LASTMOUSEX ", "LASTMOUSEY ")")
     (BLOCK)
     (GETMOUSESTATE)))
```

This function calls GETMOUSESTATE and then prints the new values of LASTMOUSEX and LASTMOUSEY in the promptwindow. To use it, type

```
(WINDOWPROP EXAMPLE.WINDOW 'CURSORMOVEDFN 'PRINTCOORDS)
```

The window property CURSORMOVEDFN, used in this example, will evaluate the function PRINTCOORDS each time the cursor is moved when it is inside the window. The position coordinates of the mouse cursor will appear in the prompt window. (See Figure 23.1.)

Figure 23.1. Current Position Coordinates of Mouse Cursor in Prompt Window

Advising GETMOUSESTATE

For the bitmap to follow the moving mouse cursor, the function GETMOUSESTATE is advised. When you advise a function, you can add new commands to the function without knowing how it is actually implemented. The syntax for advise is

(ADVISE *fn when where what*)

fn is the name of the function to be augmented. *when* and *where* are optional arguments. *when* specifies whether the change should be made before, after, or around the body of the function. The values expected are BEFORE, AFTER, or AROUND.

what specifies the additional code.

In the example, the additional code, *what*, moves the bitmap to the position of the mouse cursor. The function GETNOUSESTATE will be ADVISEd when the mouse moves into the window. This will cause the bitmap to follow the mouse cursor. ADVISE will be undone when the mouse leaves the window or when a mouse button is pushed. The ADVISEing will be done and undone by changing the CURSORINFN, CURSOROUTFN, and BUTTONEVENTFN for the window.

Changing the Cursor

One last part of the example, to give the impression that a bitmap is dragged around a window, the original cursor should disappear. Try typing:

```
(CURSOR (CURSORCREATE (BITMAPCREATE 1 l) 1 1]
```

into the Executive Window. This causes the original cursor to disappear. It reappears when you type

```
(CURSOR T)
```

When the cursor is invisible, and the bitmap moves as the cursor moves, the illusion is given that the bitmap is dragged around the window.

Functions for Tracing the Cursor

To actually have a bitmap trace (follow) the cursor, the environment must be set up so that when the cursor enters the tracing region the trace is turned on, and when the cursor leaves the tracing region the trace is turned off. The function Establish/Trace/Data will do this. Type it in as it appears (include comments that will help you remember what the function does).

```
(DEFINEQ (Establish/Trace/Data
    [LAMBDA (wnd tracebitmap cursor/rightoffset cursor/heightoffset
      GCGAGP)

      (* * This function is called to establish the data to trace
      the desired bitmap. "wnd" is the window in which the tracing
      is to take place, "tracebitmap" is the tracing bitmap,
      "cursor/rightoffset" and "cursor/heightoffset" are integers
      which detemine the hotspot of the tracing bitmap.
      As "cursor/heightoffset and "cursor/rightoffset" increase
      the cursor hotspot moves up and to the right.
      If GCGAGP is non-NIL, GCGAG will be disabled.)


(PROG NIL

      (if (OR (NULL wnd)
              (NULL tracebitmap))
          then (PLAYTUNE (LIST (CONS 1000 4000)))
               (RETURN))
      (if GCGAGP
          then (GCGAG))

        (* * Create a blank cursor.)

      (SETQ *BLANKCURSOR*(BITMAPCREATE 16 16))
      (SETQ *BLANKTRACECURSOR*(CURSORCREATE *BLANKCURSOR*))
```

```
    (* * Set the CURSOR IN and OUT FNS for wnd to the
    following:)

  (WINDOWPROP wnd (QUOTE CURSORINFN)
              (FUNCTION SETUP/TRACE))
  (WINDOWPROP wnd (QUOTE CURSOROUTFN)
              (FUNCTION UNTRACE/CURSOR))

    (* * To allow the bitmap to be set down in the window by
     pressing a mouse button, include this line.
     Otherwise, it is not needed)

  (WINDOWPROP wnd (QUOTE BUTTONEVENTFN)
              (FUNCTION PLACE/BITMAP/IN/WINDOW))
  (WINDOWPROP wnd (QUOTE CURSOROUTFN)

    (* * Set up Global Variables for the tracing operation)

  (SETQ *TRACEBITMAP* tracebitmap
  (SETQ *RIGHTTRACE/OFFSET*(OR cursor/rightoffset 0))
  (SETQ *HEIGHTTRACE/OFFSET*(OR cxursor heightoffset 0))
  (SETQ *OLDBITMAPPOSITION*(BITMAPCREATE (BITMAPWIDTH
     tracebitmap)
                                        (BITMAPHEIGHT
     tracebitmap)))
  (SETQ *TRACEWINDOW* wnd]))
```

When the function Establish/Trace/Data is called, the functions SETUP/TRACE and UNTRACE/CURSOR will be installed as the values of the window's WlNDOWPROPS, and will be used to turn the trace on and off. Those functions should be typed in, then:

```
(DEFINEQ (SETUP/TRACE
    [LAMBDA (wnd)

      (* * This function is wnd's CURSORINFN.
      It simply resets the last trace position and the current
      tracing region.  It also readvises GETMOUSESTATE to perform
      the trace function after each call.)

    (if *TRACEBITMAP*
      then (SETQ *LAST-TRACE-XPOS* -2000)
          (SETQ *LAST-TRACE-YPOS* -2000)
          (SETQ *WNDREGION* (WINDOWPROP wnd (QUOTE REGION)))
          (WINDOWPROP wnd (QUOTE TRACING)
              T)

      (* * make the cursor disappear)

      (CURSOR *BLANKTRACECURSOR*)
      (ADVISE (QUOTE GETMOUSESTATE)
            (QUOTE AFTER)
            NIL
            (QUOTE (TRACE/CURSOR]))

(DEFINEQ (UNTRACE/CURSOR
 [LAMBDA (wnd)

      (* * This function is wnd's CURSOROUTFN. The function first
      checks if the cursor is currently being traced;  if so, it
      replaces the tracing bitmap with what is under it and then
      turns tracing off by unadvising GETMOUSESTATE and setting the
      TRACING window property of *TRACEWINDOW* to NIL.)

 (if (WINDOWPROP *TRACEWINDOW*(QUOTE TRACING))
```

```
      then (BITBLT *OLDBITMAPPOSITION* 0 0 (SCREENBITMAP)
               (IPLUS (CAR *WNDREGION*)*LAST-TRACE-XPOS*)
               (IPLUS (CADR *WNDREGION*)*LAST-TRACE-YPOS*))
           (WINDOWPROP *TRACEWINDOW*(QUOTE TRACING)
                                   NIL))

      (* * replace the original cursor shape)

 (CURSOR T)

      (* * unadvise GETMOUSESTATE)

 (UNADVISE (QUOTE GETMOUSESTATE]))
```

The function SETUP/TRACE has a helper function that you must also type in. It is TRACE/CURSOR:

```
(DEFINEQ (TRACE/CURSOR
 [LAMBDA NIL

      (* * This function does the actual BITBLTing of the tracing
      bitmap.  This function is called after a GETMOUSESTATE, while
      tracing.)

 (PROG ((xpos (IDIFFERENCE (LASTMOUSEX *TRACEWINDOW*)
              *RIGHTTRACE/OFFSET*))

        (ypos (IDIFFERENCE (LASTMOUSEY *TRACEWINDOW*)
              *HEIGHTTRACE/OFFSET*))

      (* * If there is an error in the function, press the right
      button to unadvise the function.  This will keep the machine
      from locking up.)

      (if (LASTMOUSESTATE RIGHT)
          then (UNADVISE (QUOTE GETMOUSESTATE)))
      (if (AND (NEQ xpos *LAST-TRACE-XPOS*)
               (NEQ ypos *LAST-TRACE-YPOS*))
          then

      (* * Restore what was under the old position of the trace
      bitmap)

          (BITBLT *OLDBITMAPPOSITION* 0 0 (SCREENBITMAP)
              (IPLUS (CAR *WNDREGION*)*LAST-TRACE-XPOS*)
              (IPLUS (CADR *WNDREGION*)*LAST-TRACE-YPOS*))

      (* * Save what will be under the position of the new trace
      bitmap)

          (BITBLT (SCREENBITMAP)
              (IPLUS (CAR *WNDREGION*)
                  xpos)
              (IPLUS (CADR *WNDREGION*)
                  ypos)*OLDBITMAPPOSITION* 0 0)

      (* * BITBLT the trace bitmap onto the new position of the
      mouse)

          (BITBLT *TRACEBITMAP* 0 0 (SCREENBITMAP)
              (IPLUS (CAR *WNDREGION*)
                  xpos)
              (IPLUS (CADR *WNDREGION*)
                  ypos)
              NIL NIL (QUOTE INPUT)
              (QUOTE PAINT))

      (* * Save the current position as the last trace position.)

          (SETQ *LAST-TRACE-XPOS* xpos)
```

```
(SETQ *LAST-TRACE-YPOS* ypos]))
```

The helper function for UNTRACE/CURSOR, called UNDO/TRACE/DATA, must also be added to the environment:

```
(DEFINEQ (UNDO/TRACE/DATA
    [LAMBDA NIL

           (* * The purpose of this function is to turn tracing off
            and to free up the global variables used to trace the
            bitmap so that they can be garbage collected.)

            (* * Check if the cursor is currently being traced.
            It so, turn it off.)

      (UNTRACE/CURSOR)
      (WINDOWPROP *TRACEWINDOW*(QUOTE CURSORINFN)
                  NIL)
      (WINDOWPROP *TRACEWINDOW*(QUOTE CURSOROUTFN)
                  NIL)
      (SETQ *TRACEBITMAP* NIL)
      (SETQ *RIGHTTRACE/OFFSET* NIL)
      (SETQ *HEIGHTTRACE/OFFSET* NIL)
      (SETQ *OLDBITMAPPOSITION* NIL)
      (SETQ *TRACEWINDOW* NIL)

           (* * Turn GCGAG on)

      (GCGAG T]))
```

Finally, if you included the WlNDOWPROP to allow the user to place the bitmap in the window by pressing a mouse button, you must also type this function:

```
(DEFINEQ (PLACE/BITMAP/IN/WINDOW
    [LAMBDA (wnd)

      (UNADVISE (GETMOUSESTATE))
      (BITBLT *TRACEBITMAP* 0 0 (SCREENBITMAP)
           (IPLUS (CAR *WNDREGION*)
                xpos)
           (IPLUS (CADR *WNDREGION*)
                ypos)
           NIL NIL (QUOTE INPUT)
           (QUOTE PAINT]
```

That’s all the functions!

Running the Functions

To run the functions you just typed in, first set a variable to a window by typing something like

```
(SETQ EXAMPLE.WINDOW (CREATEW))
```

into the Executive Window, and sweeping out a new window. Now, set a variable to a bitmap, by typing, perhaps,

```
(SETQ EXAMPLE.BTM (EDITBM))
```

Type

```
(Establish/Trace/Data EXAMPLE.WINDOW EXAMPLE.BTM))
```

When you move the cursor into the window, the cursor will drag the bitmap.

(If you want to be able to make menu selections while tracing the cursor, make sure that the hotspot of the cursor is set to the extreme right of the bitmap. Otherwise, the menu will be destroyed by the BITBLTs of the trace functions.)

To stop tracing, do one of the following:

- Move the mouse cursor out of the window
- Press the right mouse button
- Call the function UNTRACE/CURSOR

24. GLOSSARY OF GLOBAL SYSTEM VARIABLES

As you can tell by now, there are many system variables in Medley that are useful to know. The following sections gather many of the important variables together into groups relating to directory searching, system flags, history lists, system menus, windows, and, of course, the catchall miscellaneous category.

Directories

DISPLAYFONTDIRECTORIES

Its value is a list of directories to search for the bitmap files for display fonts. Usually, it contains the FONT directory where you copies the bitmap files (see Chapter 16), and the current connected directory. The current connected directory is specified by the atom NIL. Here is an example value of DISPLAYFONTDIRECTORIES.

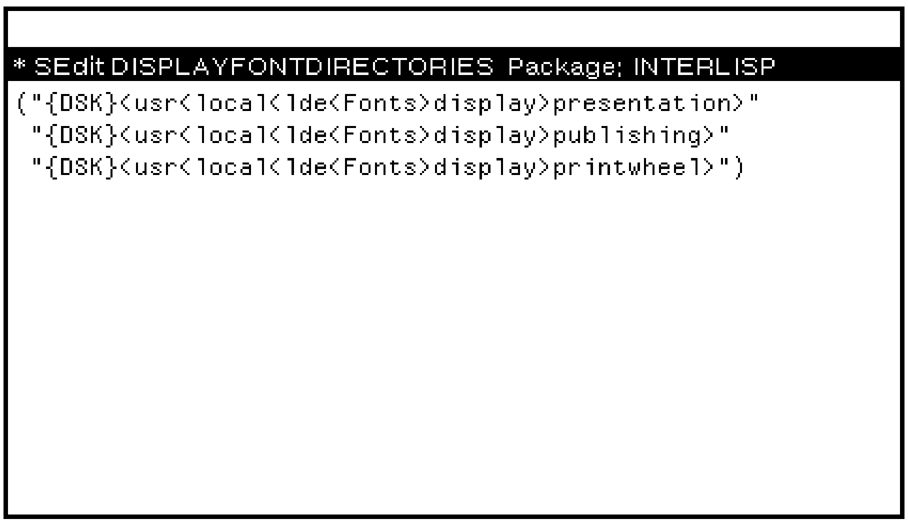


Figure 24.1. Value for the Atom DISPLAYFONTDIRECTORIES

INTERPRESSFONTDIRECTORIES

Is set to a list of directories to search for the font width files for InterPress fonts.

DIRECTORIES

This variable is bound to a list of the directories you will be using (see Figure 24-2). The system uses this variable when it is trying to find a file to load. It checks each directory in the list, until the file is found. NIL in list means to check the current connected directory.

LISPUSERSDIRECTORIES

Its value is a list of directories to search for library package files.

Flags

DWIMIFYCOMPFLG

This flag, if set to T, will cause all expressions to be completely dwinified before the expression is compiled (see Chapter 9). In this state, when the system does not recognize a function of keyword, it will compare the word to a system maintained list to determine whether the word is a macro, CLISP word, or misspelled user-defined variable.

An example of swinifying before compilation is to convert an IF call to a COND. before they are compiled. Undwimified expressions can cause inaccurate compilation. This flag is set by the system to NIL. Normally, you want this set to T. For more information on DWIM, refer to the *IRM*.

SYSPRETTYFLAG

When set to T, all lists returned to the executive window are pretty printed. This flag is originally set by the system to NIL.

CLISPIFTRANFLG

When set to T, keeps the IF expression, rather than the COND translation in your code.

PRETTYTABFLG

When set to T, the pretty printer puts out a tab character rather than several spaces to try to make code align. If NIL, it uses space characters instead.

FONTCHANGEFLG

If NIL, then when pretty printing no font changes will happen (e.g., a smaller font for comments, bold for clip words, and so forth). The default is the atom ALL, so different fonts are used where appropriate.

AUTOBACKTRACEFLG

There are many possible values for this variable. They affect when the back trace window appears with the break window, and how much detail is included in it. The values of this variable include:

- NIL, its initial value. The back trace window is not brought up when an error is generated, until you open it yourself.
- T, which means that the back trace BT window is opened for error breaks
- BT! brings up a back trace window with more detail, BT!, window for error breaks
- ALWAYS brings up a backtrace BT window for both error breaks, and breaks caused by calling the function BREAK
- ALWAYS! brings up a backtrace window with more detail, BT!, for both error breaks and breaks caused by calling the function

NOSPELLFLG

Is initially bound to NIL, so that DWIM tries to correct all spelling errors, whether they are in a form you just typed in or within a function being run. If the variable is T, then no spelling correction is performed. This variable is automatically reset to T when you are compiling a file. If it has some other non-NIL value, then spelling correction is only performed on type-in.

History Lists

LISPXHISTORY

Originally set to the list (NIL 0 30 100), with the following argument interpretation. The NIL is the list (implemented as a circular queue) to which the top level commands append. 0 is the current prompt number. 30 is the maximum length of the history list. 100 is the highest number used as a prompt. This is a system maintained list used by the programmers assistant commands REDO, UNDO, FIX, and ?? use to retrieve past function calls.

To delete the history list, reset the variable LISPXHISTORY to its original value of (NIL 0 30 100).

Setting this variable to NIL disables all the programmers assistant features.

EDITHISTORY

This is also set to (NIL 0 30 100), and has the same description as LISPXHISTORY. This list allows you to UNDO edits. You reset this the same way as LISPXHISTORY.

System Menus

System menus are all bound to global varieables and are easy to modyfy. If the menu name is set to the NIL value, the menu will be recreated using an items list bound to a global variable.

To change a system menu, edit the items list bound to the appropriate global variable (system menus use this items list with the default WHENSELECTEDFN), then set the value of the name to NIL. The next time you need the menu, it will be created from the items list you just edited. The names of system menus and the items lists follow.

BackgroundMenu

This is the variable bound to the menu this displays when you press the right button in the grey background area of the screen.

BackgroundMenuCommands

This list is used for the list of ITEMS for the background menu when it is created.

WindowMenu

This is the variable bound to the default window menu displayed when the right mouse button is pressed inside of a window.

WindowMenuCommands

This is the list of ITEMS for the WindowMenu.

BreakMenu

The menu displayed when the middle mouse button is pressed in a break window.

BreakMenuCommands

The list of ITEM for the BreakMenu.

Windows

PROMPTWINDOW

Global name of the prompt window.

T

Although the value T has several meanings (such as universal TRUE), it also stands for the standard output stream. As this is usually the executive window, it may be used as the name for the TTY Window at the top level. Mouse processes have their own TTY Windows. A reference to the window T in a mouse driven function (e.g., a WHENSELECTEFN, Chapter 12) will open a TTY Window for Mouse.

Miscellaneous

CLEANUPOPTION

This is a list of options that you set to automate clean-up after a work session. Example options are listing files, or recompilation. You will want to keep this set to NIL until you become comfortable with the machine.

FILELST

The list of all the files you loaded.

SYSFILES

The list of all the files you loaded for the SYSOUT file.

INITIALS

An atom you can bind to your name. If bound, the editor will add your name, in addition to the date, in the editor comment at the beginning of each function.

FIRSTNAME

If this variable is set, the system will use it to greet you personally when you log on to your machine.

INITIALSLST

A list of elements of the form (USERNAME . INITIALS) or (USERNAME FIRSTNAME INITIALS). This list is used by the function GREET to set your INITIALS, and your FIRSTNAME when you log in.

#CAREFULCOLUMNS

An integer. PRETTYPRINT estimates the number of characters in an atom, instead of computing it, for efficiency. Unfortunately, for very long atom names, errors can occur. #CAREFULCOLUMNS is the number of columns from the right within which PRETTYPRINT should compute the number of characters in each atom, to prevent these errors. Initially this is set to zero. PRETTYPRINT never computes the number of characters in an atom. If you set it to 20 or 30, when PRETTYPRINT comes within 20 or 30 columns of the right of the window, it will begin computing exactly how many characters are in each atom. This will prevent errors.

DWIMWAIT

Bound to the number of seconds DWIM should wait before it uses the default response, FIXSPELLDEFAULT, to answer its question.

FIXSPELLDEFAULT

Bound to either Y or N. Its value is used as the default answer to questions asked by DWIM that you don't answer in DWIMWAIT seconds. It is initially bound to Y, but is rebound to N when DWIMIFYing.

\TimeZoneComp

This is the global variable set to the absolute value of the time offset from Greenwich. For EST, \TimeZoneComp should be set to 5.

25. OTHER USEFUL REFERENCES

Here are some references to works that will be useful to you in addition to this primer. Some of these you have already been referred to, such as:

- The Interlisp-D Reference Manual (IRM)
- The Library Packages Manual
- The User's Guide to SKETCH

In addition, you can learn more about Lisp with the books:

- ***Interlisp-D: The languago and its usage*** by Steven H. Kaisler. This book was published in 1986 by John Wiley and Sons, NY.
- ***Essential LISP*** by John Anderson, Albert Corbett, and Brian Reiser. This book was published in 1986 by Addison Wesley Publishing Company, Reading, MA. It was informed by research on how beginners learn LISP.
- ***The Little Lisper*** by Daniel P. Friedman and Matthias Felleisen. The second edition of this book was published in 1986 by SRA Associates, Chicago. This book is a deceptively simple introduction to recursive programming and the flexible data structures provided by LISP.
- ***LISP*** by Patrick Winston and Berthold Horn. The second edition of this book was published in 1985 by the Addison Wesley Publishing Company, Reading, MA.
- ***LISP: A Gentle Introduction to Symbolic Computation*** by David S. Touretzky. This book was published in 1984 by the Harper and Row Publishing Company, NY.

Finally, there are three articles about the Interlisp Programming environment:

- Power Tools For Programmers byBeauSheil. It appeared in *Datamation* in February, 1983, Pages 131 - 144.
- The Interlisp Programming Environment by Warren Teitelman and Larry Masinter. It appeared in April, 1981, in *IEEE Computer*, Volume 14:1, Pages 25 - 34.
- Programming In an Interactive Environment, the LISP Experience by Erik Sandewall. It appeared in March, 1978, in the *ACM Computing Surveys*, Volume 10:1, pages 35 - 71.

Each of these articles was reprinted in the book ***Interactive Prog ramming Environments*** by David R. Barstow, Howard E. Shrobe, and Erik Sandewail. This book was published in 1984 by McGraw Hill, NY. The first article can be found on pages 19 - 30, the second on pages 83 - 96, and the third on pages 31 - 80.

TABLEOFCONTENTS

~)19 ~‘~

- - - -

. - - - - . - - , -

TABLE OF CONTENtr

YA8~ OF cottnritt

----- Next Message -----

Date: 19 Dec 91 14:18 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.141853pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11640>; Thu, 19 Dec 1991 14:19:05 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 14:18:53 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

TABLE OF CONTENff

TA.G TAaU0FC0NTENff

PREFACE

it wos dawn and the locd told him it was down the road a p;ece, lefl &t the hst fishing bridge in the counvy right at the apple tree stump, and onto the d;rt roadjust before the hill. At m;dnight he knew he was lo$t.
-Anonymous

Welcome to the Interlisp-D programming environment! The Interlis~D environment truly must be one of the most sophisticated and powerful tools in use by human beings. Overall, it is flexible, well thought out, and full of pleasant surprises: "Wow, here are exactly the set of functions l thought I’d need to write." Unfortunately, along with the power comes mind-numbing complexity. The Intedisp Refertnce Manual describes the functions and some of the tools available in the Interlisp-D environment. To do this takes three large volumes. Other volumes are needed to document the library packages and other newly written tools. Needless to say, it is very difficult to learn such a huge amount of material when there is no way to determine where to rtart!

We developed this primer to provide a starting point for new lnterlis~D usert, to enhanceø your excitement and challenge you with the potential before you. We assume you know a little about LISP, most likely received from taking a survey courte in Artificial Intelligence (Al), and have seen a demonrtration of how lnterlis~D runs on your 1186 or 1108. We further assume that your machine is not on a network system with a file server - though this is addressed, and that you will be working from floppy disks and the hard disk that is part of the machine. If this describes your situation, you are ready to sit down in front of your machine and follow the ste~by-step examples provided in this primer.

The primer is broken into many small chapters, and these chapters are organized into five parts. You may want to read

Parts 1 through 3 straight through, since they describe the basics of using the machine. Each chapter in Sections 4 and 5, however, can be used to learn a specific skill whenever you are ready to for it

Part one, "Introduction", includes Chapters 1 and 2. Part two, "Getting Into/Out of Interlisp", includes Chapters 3 through 5. Part three, "The lnterlis~D language and Programming Environment", includes Chapters 6 through IS. These chapters discuss primary elements in lnterIis~D, and orient you in relation to those elements. Part four, "Important Other Things to Know to Work Successfully", includes Chapters 16 through 31. Part five, "More Language and Environment and Packages", includes Chapters 32 through 44.

PREFAcE v

----- Next Message -----

Date: 19 Dec 91 14:20 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.142054pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11636>; Thu, 19 Dec 1991 14:21:05 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 14:20:54 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

PREFACE

Through out we make reference to the lnterlis~D Reference Manual by section and page number. The material in the primer is just an introduction. When you need more depth use the detailed treatment provided in the manual.

While only you can plot your ultimate destination, you will flnd this primer indispensable for clearly defining and guiding you to the first landmarks on your way.

Acknowledgements The early inspiration and model for this primer came from the Intelligent Tutoring Systems group and the Learning Research and Development Center at the University of Pittsburgh. We gratefully acknowledge their pioneering contribution to more effective artificial intelligence.

This primer was developed by Computer Possibilities, a company committed to making Al technology available. Primary development and writing was done by Cynthia Cosic, with technical writing support provided by Sam Zordich.

At Xerox Artificial Intelligence Systems, John Vittal managed and directed the project. Substantial assistance was provided by many members of the AlS staff who provided both editorial and systems support.

PREFA~ ø1

----- Next Message -----

Date: 19 Dec 91 14:33 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.143340pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11653>; Thu, 19 Dec 1991 14:33:46 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 14:33:40 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

1. ABRlEFGLOSSARY

The following definitions will acquaint you with general terms used throughout this primer. You will probably want to read through them now, and use this chapter as a reference while you read through the rest of the primer.

advising An lnterlis~D facility for specifying function modifications without necessarily knowing how a particular function works or even what it does. Even system functions can be changed with advising.

argument An argument is a piece of information given to an lnterlis~D function so that it can execute successfully. When a function is explained in the primer, the arguments that it requires will also be given. Arguments are also called Parametert.

atom The smallest rtrvcture in Lisp; like a variable in other programming languages, but can also have a property lirt and a function definition.

Background Menu The menu that appears when the mouse is not in any window and the right mouse button is pressed. A typical background menu is shown in Figure I.I.

Loops Icon
FileB’owser

Figuro 1.1. The Menu that appeort when the mouse is not in any window, and the right mouse button is pressed. Your background menu may have some different items in it

binding The value of a variable. It could be either a local or a global variable. See unbound.

bitmap A rectangular array of ‘ø pixels, ‘ø each of which is on or off representing one point in the bitmap image.

BREAK An Interlisp function that causes a function to stop executing, open a Break window, and allow the user to find out what is happening while the function is halted.

Break Window A window that opens when an error is encountered while running your program (i.e., when your program has broken). There are tools to help you debug your program from this window. This is explained further in Chapter 14, Page 14.1.
browse To examine a data strvcture by use of a display that allows the user to "move" around within the data rtructure.
button

A BRIEF GLOssAny

(1) (n.) Akeyonamouse.

(2) (v.t.) To depress one of the mouse keys when making a selection.
EAR A function that returns the head or firrt element of a list. See
CDR.

caret The small blinking arrowhead that marks where tert will appear
when it is typed in from the keyboard. An example of the caret
in the lnterIis~D Executive Window is shown in Figure 1.2.

NIL

B6+(PLus 3A

Figurø lJ. Me caret is to the right of the numher 3. When a characters frped
atthe keylsoard. it will ap~ar at the caret

CDR A function that returns the tail (that is, everything but the first
element) ofa list SeeEAR.

CLlSP A mechanism for augmenting the standard Lisp syntax. One such
augmentation included in Interlisp is the iterative rtatement.
SeeSection 13.1.

cr Please press your carriage return key.
datatype

(1) The kind of a datum. In Interlisp, there are many SystemAefine~
datatypes e.g. Floating Point, Integer, Atom, etc.
(2) A datatype can also be user~efined. In this case it is like a record
made up from system types and other user-defined datatypes.
DWlM D~whatøl-mean. Many errors made by Interlisp users could be
corrected without any information about the purpose of the
program or expression in question (e.g. misspellings, certain
kinds of parenthesis errort). The DWlM facility is called
automatically whenever an error occurt in the evaluation of an
Interlisp expression. If DWlM is able to make a correction, the
computation continues as though no error had occurred;
otherwise, the standard error mechanism is invoked.
error Occasionally, while a program is running, an error may occur
which will stop the computation. Interlisp provides ertensive
facilities for detecting and handling error conditions, to enable
the testing, debugging, and revising of impertect programs.
evaluate or EVAL Means to flnd the value of a form. For example, if the variable X
is bound to 5, we get 5 by evaluating X. Evaluation of a Interlisp
function involves evaluating thø arguments and then applying
thøfunction.

file packagø A set of functions and convøntions that facilitato thø
bookkeøping involved with working in i largø systøm consisting
of many sourcø codø files and thøir compiled countørparts.

Essentially, thø fllø packagø k:ps track of whørø things arø and

whet things hevo chonged. N 4150 kaps trøck of which files hove been modifiod end n#d to be updetod end recompiled.
form Another wey of seying ~xpre5sion. An Jntorlisp-D ex~on tbetcen be evaluated.

function A Lisp function is e piece of l;sp code thet executes end returns e veiue.

history The progremmerøs essistent is l,uilt eround e memory structure celled the hirtory Iirt. The hirtory functions (e.g. FIX, UNDO, REDO) ere part of this essirtant. These operations allow you to conveniently r~work previously specifiecl operations.

History List As you type on the xreen, you will notice a number followecl by a prompt attow. Each number, and the information on that line, is seeluentially rtored as the History List Using the History List, you can easily reexecute lines typed earlier in a worksession. See Chapter6.

icon A pictorial representation, usually of shrunken window.

lnterlis~D Executive Window This is your main window, where you will run functions and develop your programs. See Figure 1.3. This is the window that the caret is in when you turn on your machine and load lnterlis~D.

NIL

8~#iPRO*PTPRIHT "HELLO" A

Fqurø tJ. m window

inspector An interactive display program for examining and changing the parts of a data structure. Jnterlisp-D has inspectors for lists and other data types.

iterative statement (also called i.s.) A statement in Interlisp that repetitively executes abody of code. (E.g.(forxfromltosdo(PRlNTx))isani.s.)
iterative variable (also called i.v.) Usually, an iterative statement is controlled by the value that the i.v. takes on. In the iterative statement example above,
x

is the iterative variable leecause its value is being changed by each cycle through the loop. All iterative variables are local to the iterative rtatement where they are defined.

LISP Family of languages invented for "list processing." These languages have in common a set of basic primitives for creating and manipulating symbol rtructures. lnterlis~D is an implementation of the LISP language together with an environment (set of tools) for programming, an a set of packages that ertend the functionality of the system.

list A collection of atoms and lists; a list is denoted by surrounding its contents with a pair of parentheses.

Loading LJSP This is the process of bringing lnterlis~D from floppy disks, hard disks, or some other secondary rtorage into your main, or working, memory. You will need to load (i.e., install, and boot) lnterlis~D if you have not logged off the machine at the end of a session. The process of loading lnterlis~D is explained in Chapter3.

Maintenance Panel Codes Should you have a problem with your equipment, these codes will indicate the status of your processor. On the 1108, these are the red LED numbett under the floppy drive door. There is a cover over these numbers. Pull down the cover located immediately under the floppy door button. The code numbers are defined for the 1108 in the 1108 Useri Guide, in the MP Codeschapter.

If there is a problem with the 1186, the mouse curtor will change from its normal arrow to the code number that describes the problem. The code numbert are defined for the 1186 in the 1166 User’s Guide in the Curtor Codes subsection of the Diagnostics Chapter.

Marterscope A program analysis tool. When told to analyze a program, Masterscope creates a data base of information about the program. In particular, Marterscope knows which functions call other functions and which functions use which variables. Masterscope can then answer questions about the program and display the information with a browser.

menu A way of graphically presenting the user with a set of options. There are twO kinds of menus: p0~up menus are created when needed and disappear after an item has been selected; permanent menus remain on the screen after use.

mouse The Mouse is the box to the right of your keyboard. It controls the movement of the cursor on your screen. As you become familiar with the mouse, you will find it much quicker to use the mouse than the keyboard. See Figure 1.4. (Note: Some mice have three buttons; the button in the center is known as the middle mouse button. If your mouse has only two buttons, you can simulate a middle button by pressing the left and right buttons simultaneously.).

Fw- 1.& Mousø

Mouse Curtor The small arrow on the screen that points to the northwest. See Figure 1.5.

F~m I.L Mousø c~~~
Mousø Curtor Icons

A IRlEF GLOSSARY

I Wait Tho processor is busy.

The processor is saving a anpashot or your cureent system session. This is usually donø when tbc procffssor hos ~n idle for a while.

The "Mouse Confirm Cursor". It appeatt when you have to confirm that the choice you just made was correct. If it was, press the left button. If the choice was not tight, press the right button to abort.

F=’*x This means "sweep out" the shape of the window. To do this, move the mouse to a position where you want a corner. Press the leff mouse button, and hold it down. Move the mouse diagonally to sketch a rectangle. When the rectangle is the desired size and shape, release the left button.

r-’l
l

l

l
l

- This is the "move window" prompt. Move the mouse so that the large "ghost" rectangle is in the position where you want the window. When you click the left mouse button, the window will appear at this new location.

NIL NIL is the lnterlis~D symbol for the empty list h can also be represented by a lefl paren followed by a right paren: (). It is the only expression in lnterIis~D that is both an atom and a list
pixel Pixel rtands for PIcture Element. The xreen of your Lisp Machine is made up of a rectangular array of pixels. Each pixel corresponds to one bit When a bit is turned on, i.e. set to 1, the pixel on the screen represented by this bit is black.

pretty printing Pretty printing refers to the way lnterIis~D functions are printed with special indentation, to make them easier to read. Functions are pretty printed in the structure eclitor, DEdit (See Section 11.3, Page 11.4). You can pretty print uncompiled functions by calling the function PP with the function you would like to see as an argument, i.e. (PP tunction-name). For an example of this, see Figure 1.6.

96.(PP HEAD)

[LANBDA (LST) <łø6rtøG; ‘ø~JlinH13;3&ø)
(CAR LsTJ)
(HEAD)
97.’

Fbm 1.6. An oxam~ u~oftho pro printing ‘unmon FP

A BRIEF GLOSSARY

Programmer’s Assistant The programmer’s assistant accesses the History List to allow you to FIX, UlOO, and’or REDO your previous expressions typed to the lnterlis~D executive window. (See Chapter 6.)

Promptwindow The skinny black window at the top of the xreen. It displays system prompts, or prompts you have developed. (See Figure 1.7.)

Fqurø 1.7. Prompt window

property list A list of the form (<property-namel> <property-valuel> <property-name2> <property-value2> ....) associated with an atom. It accessed by the functions GETPROP and PUTPR0P.

record A record is a data-structure that consists of named "flelds". Accessing elements of a record can be separated from the details of how the data structure is actually stored. This eliminates many programming details. A record definition establishes a record template, describing the form of a record. A record instance is an actual record storing data according to a particular record template. (See datatype, second definition.)

Right Button Default Window Menu This is the menu that appeaff when the mouse is in a window, and the right mouse button is pressed. It looks like the menu in Figure 1.8. If this menu does not appear when you depress the right button of the mouse and the mouse is in the window, move the mouse so that it is pointing to the title bar of the window, and press the right button.

Clone
Snap
Paint
Clear
8ury

Redi~play
Hardcopy~
Move
Shapo
Shrink

f~a 1.1. ttø Right Sutton DqfaultWindow Menu

5-expression Short for "symbolic expression." In Lisp, this refers to any well-formed collection of leff parffns, atoms, and right parens.

stack A pushdown lirt Whenever a function is entered, information about that specific function call is pushed onto (i.e. added to the front ofi the stack; this information includes the variable names and their values associated with the function call. When the function is exitted, that data is popped off the stack.

storagø devices Information is rtored for your Lisp machine on floppy disks, or on the hard disk. They are referred to as (FLOPPY) and (DSK) respectivøly.

sysout A fllø containing G wl’0lø Lisp environmønt: namely, lnterlis~O, evørything thø user’ døfinecl or loaded into the environment, thø

A BRIEF GLOSSARY

windows that øppeored on tho Kreen, tbø ømount of memory used, and 50 on. Evorything ;s rtorød in thø sysout filø exactly .5 it was whon tho function SYSWT was called).

TFACE A function that crøates a trace of the execution of another function. Eich time the traced function is called, it prints out the values of the arguments it was called with, and prints out the value it returns upon completion.

Unbound Without value; an atom is unbound if a value has never been assigned to it

window A rectangular area of the screen that acts as the main display area for some Interlisp process,

A BRIEF GLOSSARY 1.7
1

----- Next Message -----

Date: 19 Dec 91 14:42 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.144256pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11642>; Thu, 19 Dec 1991 14:43:07 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 14:42:56 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

6. TYPING SHORTCUTS

Once you have logged it, as per Chapters 3 or 4, you are in lnterlis~D. The functions you type into the Interlisp-D executive window will now execute, that is, perform the designated task. Please note that Interlisp-D is case-sensitive; offen it matters whether text is typed in capital- or lower-case letters. The shifflock key is above the left shift key; when it is pressed (on the 1186, the red LED will be on; on the 1108, the key will be depressed), everything typed is in capital letters.
You must type all Interlisp-D functions in parentheses. The lnterlis~D interpreter wtll read from the leff parenthesis to the closing right parenthesis to determine both the function you want to execute, and the arguments to that function. Executing this function is called evaluation. When the function is evaluated it returns a value, which is then printed in the lnterlis~D executive window. This entire process is called the read-eval-print loop, and is how most Ll5P interpreters, including the one for lnterlis~D, run.

The prompt in Interlis~D is a number followed by a left pointing arrow (see Figure 6.3). This number is the function's position on the History List -- a list that stores your interactions with the lnterlis~D interpreter. Type the function (PLUS 3 4), and notice the number the History List assigns to the function (the number immediately to the leff of the arrow). lnterlis~D reads in the function and its arguments, evaluates the function, then prints the number 7.

In addition to this read-eval-print loop, there is also a programmer's assistantøø. It is the programmer's assistant that prints the number as part of the prompt in the lnterlis~D executive window, and uses these numbers to reference the function calls typed after them.

When you issue commands to the programmer's assistant, you will not use parentheses as you do with ordinary function calls. You simply type the command, and some specification that indicates which item on the history list the command refers to. Some programmer's assistant commands are FIX, REDO, and UNDO. They are explained in detail below.

Programmer's assistant commands are useful only at the lnterlis~D top level, that is, when you are typing into the lnterlis~D executive window. They will not work in user-defined functions.

As an example use of the programmer's assistant, use REDO to redo your function call (PLUS 3 4). Type REDO (Note: programmer's assistant commands can be typed in either upper

or lower case) at the prompt, then specify the previous expression in one of the following ways:

(1) When you originally typed in tne function you now want to refer to, there was a History List number to the left of the arrow in the prompt. Type this number affer the programmer's assistant command. This is the method illustrated in the following figure:

iPLUe~ 3 4)
Cø'5~REOO '24
7

26-','

, , , . . , . , ... , ... , , ,
, .

Figure 6.1. Using the programrner's assistant to REDO a function, when you know the its number on the history list

(2) A negative number will specily the function call typed in that number of prompts ago. in this example, you would type in-I, the position immediately before the current position. This is shown in the following figure:

'2ł~.(PLUS :3 j)
;;;;

'-,7'

ii, ~' F.EDS -i
7

:ø'0~.

,
, , , , ,: , , :, , ,:, , , , , , ;, : , , , ,; , ,

Figurø 6.2. Typing a negative number affer the programmer's assistant command will cause it use the function found on the History List that many positions before the current one.

(3) You can also specify the function for the programmer's assistant with one of the items that was in that function call. The programmer's assistant will se'rch backwards in the History List, and use the first function it finds that includes that item. For example, type REDO PLUS to have the function (PLUS 3 4) reevaluated.

(4) If you type a Programmer's Assistant command without specifying a function (i.e., simply typing the command, then a cr) the Programmer's Assistant executes the command using the function entered at the previous prompt.

Here are a few more examples of using the programmer's
assistant:

G.a TYPING SHORrCUff
1

TYPING SH0RTCUTS
NIL

54k[PLUS 4 5)
9

55~REDO
9

56#??

54 +(PLUC~ 4 5)
9

56~(SETQ B '80Y)
BOY
5'~B
BOY

59" UNDO cETQ
SETQ undone.
59'.B

UN8OUND nTOM
B

SBkREDO 56
BOY
6IkB
BOY
62#

Fqøurø 6.3. Some Applications of the Programmer's Assistant

6.1 If you make a Mistake

Editing in the Interlisp-D Executive Window is explained in
Section 11.2, Page 11.2. In this section, only a few of the most
useful commands will be repeated.

To move the caret to a new place in the command being typed,
point the mouse cursor at the appropriate position, and press the
leff mouse button.

To move the caret back to the end of the command being typed,
press CONTROL-X. (Hold the CONTROL key down, and type ø.X.'.)
The way you choose to delete an error may depend on the
amount you need to remove. To delete:

The character behind the caret simply press the backspace key
The word behind the caret press CONTROL-W. (Hold the CONTROL key down, and rype
'øW'ø.)

Any part of the command, first move the caret to the appropriate place in the command.
Hold the right mouse button down and move the mouse cursor
over the ten. All of the blackened tert between the caret and
mouse cursor is deleted when you release the right mouse
button.

TYPING SHORTCUTS 63

IF YOU MAKE A MISTAKE

The entire command press CONTROL-U. (Hold the CONTROL key down, and type in".)
Deletions can be undone. Just press the UNDO key.
To add more text to the line, move the caret to the appropriate position, and just type. Whatever you type will appear at the caret.

6.4 TYPING SHORTcUTS

----- Next Message -----

Date: 19 Dec 91 14:48 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.144827pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11544>; Thu, 19 Dec 1991 14:48:38 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 14:48:27 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

7. USING MENUS

The purpose of this chapter is to show you how to use menus. Many things can be done more easily using menus, and there are many different menus provided in the Interlisp-D environment. Some are "po~up" menus, that are only available until a selection is made, then disappear until they are needed again. An example of one of these is the "background menu", that appears when the mouse is not in any window and the right mouse button is pressed. A background menu is shown in Figure 7.1. Yours may have different items in it.

SkGtL'h
LUop3 Icon
CHAT
F.lle0ro~er

sav"VM
5nap

Figure 7.1. A hackground menu.

Another common pop-up menu is the right button default window menu. This menu is explained more in Section 10.4, Page 10.3.

Other menus are more permanent, such as the menu that is always available for use with the Interlisp-D Filebrowser. This menu is shown in figure Figure 7.2, and the specifics of its use with the filebrowser is explained in Chapter 9).

Dnjelsta
Rcname
Hor~'UpJ
-=‘ffl.e

Compil'.
E~prnnge

Recrjm Ut.fl,L'

Figure 7.2. The menu that is available when using the Filehrowser

MAKING A SELEcTION FROM A MENU

7.1 Making a Selection from a Menu

To make a selection from a menu, point with the inouse to the item you would like to selert If one of the moU5e buttons is already pressed, the menu item 5hould blacken. If it is a permanent menu, you must press the leff mouse button to blacken the item. When you release the button, the item will be chosen. Figure 7.3 shows a menu with the item "Undo" chosen.

ø ø1
.lffer
Bpfor',
GeIer~,
Replace
'witch
(

'3 LIt.

Find
'=w~
pcpflnt
Edt

Edfl-Um

0~ik

Eva
E.xit

Figure 73. A menu with the item "Undo" chosen

7.2 Explanations of Men.u Items

Many menu items have explanations associated with them. If you are not sure what the consequences of choosing a particular menu item will be, blacken the menu item, and do not release the leff button. If the menu item has an explanation associated with it, the explanation will be printed in the prompt window. Figure 7.4 shows the explanation associated with the item "Snap" from the background menu.

ile0row~or

Flguvø 7.& The explanation associated with the cliosen item, Snip, is displayed in the prom pt window.

SUBMENUS

7.3 Submenus

Some menus items have submenus associated witl, them. This means that, for these items, you can make even more precise choices if you would like to.

A submenu can slso be found in one of two ways. One is to point to the item with the mouse cuttor, and press the middle mouse button. If there is a submenu associated with that item, it will appear. (See Figure 7.5.)

øl
Atter
8e?are
DoloCe
Replace
Yvitch

‘ut
l)nda
Find
cap
Repnnt
Edit

EditL’om
Break

Eva OK

TOP

FigurQ 7.5. The submenu associated with the menu item Exit It appeared when the mouse curtor Pointed to the menu item. and the middle mouse button was pressed.

A submenu can be indicated by a gray arrow to the right of the menu item, like the one to the right of the "Hardcopy’ø choice in Figure 7.1. To see the submenu, blacken the menu item, and move the mouse to follow the arrow. An example of this is shown in Figure 7.6. Choosing an item from a submenu is done in the same way as choosing an item from the menu. Any submenus that might be associated with the items in the submenu are indicated in the same way as the submenus associated with the items in the menu.

Dnclelete .~ .
Copy

Rename
Harjcopv

.=.ee ~~e~;

Loa.d c’.Ee!T.:
E,puni~e

P’,com Ll!eø

Figure 7.6. The submenu associated with the menu item Edit - It appeared when the menu item was blackened, and the mou>e was moved (0 follow the gray arrow.

In summary, here are a few rules of thumb to rerrember about the interactions of the mouse, and system menus:

ø Press the leff mouse button to select an item of a menu ø Press the middle mouse button to get more options - one of the ways to find a submenu

USING MENUS 73

SUBMENU5

ø Press the right mouse button to see the default right button window menu, and the background menu

7.4 usiNG MENUS

----- Next Message -----

Date: 19 Dec 91 14:56 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.145658pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11659>; Thu, 19 Dec 1991 14:57:09 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 14:56:58 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

8. MOW TO USE FILES

8.1 Types of Files

A program file, or lisp file, contains a series of expressions that can be read and evaluated by the lnterlis~D interpreter. These expressions can include function or macro definitions, variables and their values, properties of variables, and soon. How to save Interlisp-D expressions on these files is explained in Section 11.6, Page 11.7. Loading a file is explained below, in Section 8.6, Page 8.4.

Not all files, however, have lnterlis~D expressions stored on them. For example, TEdit files (see Chapter 23) store tert; sketches are stored on files made with the package Sketch (see Chapter 35), or can be incorporated into TEdit files. These files are not loaded directly into the environment, but are accessed with the package used to create them, such as TEdit or Sketch. When you name a file, there are conventions that you should follow. These conventions allow you to tell the type of a file by the extension to its name. If a file contains:

Interlisp-D expressions, it should not have an extension. For example, a file called "MYCODE" should contain lnterlis~D expressions;

compiled code, it should have the extension" .DCOM’ø. For example, a file called ‘øMYCODE.DCOM" should contain compiled code;

a Sketch, then its extension should be ".SKETCHøø. For example, a file called ø‘MOUNTAlNS.SKETCH" should contain a Sketch;

text, it should have the extension ".TEDlTø’. For example, a file called ‘øREPORT.TEDlT’ø should contain text that can be edited with the editor TEDlT.

8.2 Directories

This section focuses on how you can find files, and how you can easily manipulate files. To see all the files listed on a device, use the function DIR. For example, to see what files are stored on the Y;ard disk, type
(DIR (DSK))

HOW TO USE FILES B1

DlRžG0R1E5

To see what files are stored on the floppy disk inside of the floppy drive, type
(DIR (FLOPPY))

Partial directory listings can be gotten by specifying a file name, rather than just a device name. The wildcard "ø‘ø can be used to match any number of unknown characters. For example, the command
(DIR (DSK)Tø)

will list the names of all files stored on the hard disk that begin with the letter T. An example using the wildcard is shown in Figure 8.1

‘DIR ‘(P\h’,(LI’.\PFILž.’.’PRIMER~T’;l
‘LPQh/’’.LI."’l FILE.C\,’PRIMER\
Tsi’REF.>.TEP[1)2
T6LlClNT.TEDIT,1

FigurG 8.1. Using the function DIR with a wildcard

8.3 Directory Options

Various words can appear as extra arguments to the DIR command. These words give you extra information about the files.

(1) SIZE displays the size of each file in the directory. For example, type

(DIR (DSK) SIZE)

(2) DATE displays the creation date of each file in the directory. An example of this is shown in Figure 8.2

35~(DIR (DsxJ.<LI$PF1LEs>PRIMER~T* DATE)
CREATIDNOATE

(øDSK)’LI5PFILES~PRIflER?

TA’1"REF~TEPlT;2 26-lun-R5 19:A,O:R2
TBLnrnNT.TEDIT;1 26-lun’66 ja:4R~ø?

3Lq~

. . . . . . : . . . .

Figure 8.2. An eximpie using thø dirøctory option DATE
(3) DEL deletes all tho files found by the directory command

G.a H0W TO USE FILES

SUøFlLE DlREO0RlES

8.4 Subfile Directories

Subfile directories are very helpful for orgonizing files. A set of files that have a single purpose, for example all the external documentation files for a system, can be grouped together into a subfiledirectory.

To associate a subfile directory with a filename, simply include the desired subfile directory as part of the name of the fileø Subfile directories are specified after the device name and before the simple filename. The first sibfile directory should be between less-than and greater-than signs < >, with nested subdirectory names only followed by a greater-than sign >ø For example:

[DSK)<D1røctory>SubOlmctory>SrnbSubDirøctor-y>.. .>fi1on~

8.5 To See What Files Are Loaded
If you type FILELST<CR>, the names of all the files you loaded will display.

Type SYSFILES<CR>, to see what files are loaded to create the SYSOUT.

8.6 Simple Commands for Manipulating Files
The following commands will work with the (FLOPPY) and other devices, but have been shown with (DSK) for simplicityø
To have the contents of a file displayed in a window:
(SEE '[DSK)f11ønrn)

To copy a file: (coPYFILE '[~)o1dfi1øn~ '[DSF)ne,r,,ilonrn)
An example of this is shown in Figure 8.3
(sOPvFILE 'T~0r,RžFc.TEDIT 'PF;IMEFRøEFOø.TžDITJ
t'Dcxl,(.LIsPFILEs.PRIMžP.;ø.PRIMEP.fiEFs.TEDIT;1

Figure 8.3. An example of the use of the function COPYFILE
To delete a file: (DEl.FILž '(~)fi1on~)
An example of this is shown in Figure 8.4.
ø ,, OELFILž 'L'AMPLE.TEPITJ
ø . \l.. I'PfILE;'."PRIMER?>AnPLE.TžPIT;1

FigureS.O. The function DELFlLE

To rename a file: (RENlEFILE ø(osK)oldftlønrn '(rSF)ner,r11øn~)

HOW TO USE FILES 83
1

SIMPLE COMMANDS FOR MANIPULATING FILES

"LOAD" a file: Files that contain Interlisp-D expressions can be loaded into the environment. That means that the information on them is read, evaluated, and incorporated into the Interlisp-D environment.
To load a file, type:

(LUG '[DSff)filenm)

When using these functions, always be sure to specify the full filename, including subfile directories if appropriate.

8.7 Connecting to a Directory
Offen, each person or project has a subdirectory where their files are stored. If this is your situation, you will want any files you

create to be put into this directory automatically. This means you should "connect" to the directory.

CONK is the Interlisp-D form that connects you to a directory. For example, COilKin the following figure:
- 1 l 11

29#(L'OtJN ''CDv~K1,.LIv"PFILES~\PP,IMER7IM\,!,I
t'OS'Y96)cLIy'PFILžCv;~PRIh1žR.~lM>
30#

Fqrnre 8.5. COflffeaing to the subdiredory "PRIMERs srnbsu~i'edory ,.lM" connects you to the subsubdirectory iM, in the subdirectory PRIMER, in the directory LlSPFlLES, on the device D5K. This information, the device and the directory names down to the subdirectory you want to be connected to, is called the "path" to that subdirectory. co:: expects the path to a directory as an argument.

Once you are connected to a directory, the command DIR will assume that you want to see the files in that directory, or any of its subdirectories.

Other commands that require a filename as an argument (e.g., SEE, above) will assume, if there is no path specified with the filename, that the file is in the connected directory. This will often save you typing.

8.8 File Version Numbers

Whe.n stored, each file name is followed by a semicolon and a number.

fffILE.TEOIY;1

The number ii the vertion number of the file. This is the system's way of protecting your files from being overwritten. Each time the file is written, a new file is created with a vettion number one

greater than the lost. This now fle will hove everything from your previous file, plus all of your changes.

In most cases, you can exclude the version number when referencing the file. When the vertion is not specified, and there is more that one vertion of the file on that particular directory, the System generally uses your most recent version. An exception is the function DELFILE, which deletes the oldest version (the one with the lowest vertion number) if none is specified.

----- Next Message -----

Date: 19 Dec 91 15:03 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.150359pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11664>; Thu, 19 Dec 1991 15:04:10 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 15:03:59 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

tO.THOSE WONDERFUL WINDOWS!

A window is a designated area on the screen. Every rectangular box on the screen is a window. While Interlisp-D supplies many of the windows (such as the lnterlis~D executive window), you may also create your own. Among other things, you will type, draw pictures, and save portions of your screen with windows.

10.1 Windows provided by Interlisp-D
Two important windows are available as soon as you enter the lnterlis~D environment. One is the lnterlis~D executive window, the main window where you will run your functions. It is the window that the caret is in when you turn on your machine, and load lnterlis~D. It is shown in Figure 10.1.

Figure 10.1. Interlsp-D Executive Window

The other window that is open when you enter Interlisp-D is the "Prompt Window". It is the long thin black window at the top of the screen. It displays system prompts, or prompts you have associated with your programs. (See Figure 10.2.)

Figure 10.2. Prompt Window

Other programs, such as the editors, also use windows. These windows appear when the program starts to run, and close (no longer appear on the screen) when the program is done running.

THOSE WONDERFUL WINDOWS’ 101

CREATING A WINDOW

10.2 Creating a window

To create a new window, type: (CREATEil). The mouse cursor will change, and have a small square attached to it. (See Figure 10.3.)

Figure 10.3. The mouse cursor asking you to sweep out a window
There may be a prompt in the prompt window to create a window. Press and hold the leff mouse button. Move the mouse around, and notice that it sweeps out a rectangle. When the rectangle is the size that you’d like your window to be, release the leff mouse button. More specific information about the creation of windows, such as giving them titles and specifying their size and position on the xreen when they are created, is given in Section 27.1.2, Page 27.2.

10.3 The Right Button Default Window Menu
Position the cursor inside the window you just created, and press and hold the right mouse button. A menu of commands should appear (do not release the right button!), like the one in figure 10.4. To execute one of the commands on this menu, choose the item. Making a choice from a menu is explained in Section 7.1, Page 7.2.
clQ1,ø/
Pant

‘[oar
Bury

RoJisplay
Hardsopy~
Movc
‘5hape
shrink

Figurø 1O.& The Right Button Default Window Menu

As an example, select "Move" from this menu. The mouse cursor will become a ghost window Oust an outline of a window, the same size as the one you are moving), with a square attached to one corner, like the one shown in Figure 10.5.
~l

~l

F~ure 10.1 Thø mousø cunor !or moving & window

Move the mousø around. The ghost window will follow. Click the left mousø button to placø tho window in a new location.

10.1 TH0Sff w0NKQFUL WlH00~l

f

THE RIGHT 8Uff0N DEFAULT MN00~ MFNU

Choose "Shape", afid notice that you are prompted to sweep out another window. Your original window will have the shape of the window you sketch out.

10.4 An explanation of each menu item
The meaning of each right button default window menu item is explained below:

Close removes the window from the screen;
Snap copies a portion of the screen into a new window;
Paint allows drawing in a window;

Clear cleart the window by erasing everything within the window boundaries;

Bury puts the window beneath all other windows that overlap it;
Redisplay redisplays the window contents;

Hardcopy sends the contents of the window to a printer or to a flle;
Move allows the wi ndow to be moved toanew spot on the screen;
Shape repositions and/or reshapes the window;
Shrink reduces the window to a small black rectangle callecl an icon. (See Figure 1O6.)

Figum lO.L An example icon

Expand changes an icon back to iB original window. Position the mouse cursor on the icon, depress the right button, and select Expand.
Or, just button the icon with the middle mouse button.
These right-button default window menu selections are available in most windows, including the lnterlis~D Executive window. When the right button has other functions in a window (as in an editor window), the right button default window menu should be accessible by pressing the Right button

in the black border at the top of the window.

10.5 Scrollable Windows

Some windows in Interlisp-D are "scrollable". This means that you can move the contents of the window up and down, or side to side, to see anything that doesn’t fit in the window.

Point the mouse curtor to the leff or bottom border of a window. If the window is scrollable, a "scroll bar" will appear.

The mouse cursor will change to a double headed arrow. (See Figure 10.7.)

. 1 , 1

Figuro 10.7. The scrolJ bar of a scrollable window. The mouse cursor changes o a double headed arrow.

The xroll bar represents the full contents of the window. The example scroll bar is completely white because the window has nothing in it When a part of the scroll bar is shaded, the amount shaded represents the amount of the window’s contents currently shown. If everything is showing, the scroll bar will be fully shaded. (See Figure 10.8.) The position of the shading is also important. It represents the relationshi’p of the section currently diplayed to the the full contents of the window. For example, if the shaded section is at the bottom of the scroll bar, you are looking at the end of the file.

1 ø .

The amount of :>hadin~ in
A::;:. the scroll bar represents
the amount of the rile

;>hown in the window. Most
of the file is visible.
Because the shading is at
the tap of the scroll bar,
you know you are looking
at the top of the file.

Figurø 10.1 Tho amount of shading in the scroll bar represents the amount of the file shown in the window. Most of the file is visible. Because the shading sat the top of the scroll bar. you know you are looking at the top of the file

When the scroll bar is visible, you can control the section of the window’s contents displayecl:

ø To move the contents higher in the window (scroll the contents up in the window), press the leff button of the mouse, the mouse cursor changes to look like this:

Fun 10.1. upward icrollinq cuttor.

The contents of the window will scroll up, making the line thit the cursor is beside the topmost linø in the window.

SCROLLASLE MN00~S

ø To move the contonts lower in the window (scroll the contents øødown" in the window), press the right button of the mouse, ond the mouse cursor changes to look like this:

Flgrnro 10.10. Oownwørd scrollinq curtor

The contents of the window scroll down, moving the line that is the topmost line in the window to beside the curtor.
ø To show a specific section of the window's contents, remember that the scroll bar represents the full contents of the window. Move the mouse curtor to the relative position of the section you want to see (e.g., to the top of the scroll bar if you want to see the top of the window's contents.). Press the middle button of the mouse. The mouse cursor will look like this:

f'9ure 10.11. Proportional scrolling crnrtor.

When you release the middle mouse button, the window's contents at that relative position will be displayed.

10.6 Other Window Functions

10.6.1 PROMPTPRlNT

Prints an expression to the black prompt window.
For example, type

(P~PTPRIKT øTNIS SILL BE PRIKTED I* THE PAT UIKOoS')
The message will appear in the prompt window. (See Figure 10.12.)

1 . ø1 Il

43 lpROMPTPRINT 'THIS WILL BE PRINTED IN THE PROMPT WINDOW')

Flurf 10.12. PROMPTPRlNTing

THOSE WONDERFUL WINDOWS' 10.5

OTHER WINDOW FUNCTIONS

10.6.2 WHlCNW

Returns as a value the name of the window that the mouse cursor IS in.

(VHICHW) can be used as an argument to any function expecting a window, or to reclaim a window that has no name (that is not attached to some particular part of the program.).

10.6 THOil wONOERFUL~N00vn'

----- Next Message -----

Date: 19 Dec 91 15:18 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.151815pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11655>; Thu, 19 Dec 1991 15:18:21 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 15:18:15 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

tl. žDlTUNG AND SAVING

This chapter explains how to define functions, how to edit them, and how to save your work.

11.1 Defining Functions

DEFINEQ can be used to define new functions. The syntax for it is:

(HFIffEQ (<tunctionname> (<parameterlist↓
c~y-offrnnction>j>

New functions can be created with DEFINEQ by ryping directly into the lnterlis~D executive window. Once defined, a function is a part of the lnterlis~D environment. For example, the function EXANPLE-ADDER is defined in Figure 11.1.
-

HIL

46=(OEFINEQ (E.~AMPLE-rt"D&ER (~" B cJ
(PRINT ‘THE SUM OF THE
THREE NUMBERS Is ")
(IPLUS n" B CJJJ
(EXn~MPLE-~&DERj
47-

F1ure 11.1. Defining the function EXAMPLEøADDER

Now that the function is defined, it can be called from the lnterlis~D executive window:
ø . -

NIL

49’. cEX~-MPLE-ffD&ER 3 4 ‘J;
"THE SUM OF THE THREE NljMBERS 15
12

c~g

Fq’rnre IIJ. After EXAMPLžøADDER is defined, it can he executed
The function returns 12, after printing out the message.
Functions can also be defined using the editor DEdit described above. To do this, simply type

(DF furttiorvnamej

DEFINING FUNCTIONS

You will be asked whether you would like to edit a Dummy definition. A dummy definition is a standard template for your

function definition. Answer by typing Y for Yes, and you will be able to define the function in the editor. (See Figure 11.3. The use of the editor is explained in Section 11.3, Page 11.4.)
ø h.’1,flF PJž~-HOT’E’[’øTi

ø Ho FH~., dean ;or oil NflT-Eøøø 1:-7, on øoU Ui ‘;øh ro ?dlr a ‘lu

60A "Fløø;:øl

Figurn 11.1 Using DEdit to define a function

II _ 2 _ Simple _ Editing _ in _ the _ Interlisp-D _ Executive Window
First, type in an example function to edit:
51~(oEFIxEQ (Y~R-FIRST-fuKTIrn (A B)
(if (GREATERP A B

thøn TNE FIR T IS GREATER
elsø THE SECO*O IS 6REATE))))

To run the function, type (YOUR-FIR$T-FUflcTIoa 3 5).
52~(Y~R-FIRST-Fu~TI: 3 5)
(TNE SEc~ Is GREATER)
Now, let’s alter this. Type:
53~FIZ 51 cr

Notf that your original function is redisplayed, and ready to edit. (SeeFigure 11.4.)

llJ EO1Y1~ AHO SAVING

r,

SIMPLE EDITING IN TNE INTERLISPøD EXECUTIVE WIND0W

NIL

53~FI~ 51
+(DEFINEQ

[YOUR-FIRST-FUNCTION
(A B) (ø edited;

"~1-Dec-GB 19;"8")
(IF (GREaTERPøA B)
THEN (QUOTE (THE FiRST Is
UREATERj)

ELSE (QUOTE (THE SECOND IS
u’RE~~TER] 1A

f~urø11.& Using FIX to editafundion

Move the tert cursor to the appropriate place in the function by positioning the mouse cursor and pressing the Jeff mouse button.

Delete text by moving the caret to the beginning of the section to be deleted. Hold the right mouse button down and move the mouse cursor over the text. All of the blackened text between the caret and mouse cursor is deleted when you release the right mouse button.

If you make a mistake deletions can be undone. On an 1108, press the OPEN key to. UNDO the deletion. On an 1108, press the UNDO key on the

keypad to the Jeff of the keyboard.
Now changeGREATERtoBIGGER:

(1) Position the mouse cursor on the G of GREATER, and click the leff mouse button. The text curtor is now where the mouse cursor 15.

(2) Next, press the right mouse button and hold it down. Notice that if you move the mouse cursor around, it will blacken the characters from the text cursor to the mouse cursor. Move the mouse so that the word "GREATER" is blackened.
(3) Release the right mouse button and GREATER is deleted.
(4) Without moving the cursor, type in BIGGER.
(5) There are two ways to end the editing session and run the function. One is to type CONTROL-X. (Hold the CONTROL key down, and type "X".) Another is to move the text cursor to the end of the line and crø In both cases, the function has been edited!

Trythe new version of the function bytyping:
58~(Y~-FZRST-F~Tzrn 8 9)
(TNž sEc~ Is BIKER)

and get the new result, or you can type:
5~RE00 52cr

(TNE SEc~ Is BIKER)

EDITING AND SAVING 11.3

USING THE LIST STRUaURE EDITOR

11.3 Using The List Structure Editor
If the function you want to edit is not readily available (i.e. the function is not in the Interlisp-D Executive window, and you can’t remember the history list number, or you simply have a lot of editing), use the List Structure Editor, offen called DEdit. This editor is evoked with a call to OF:

81~(DF YWR-FIRST-f~TIa)

Your function will be displayed in an edit window, as in Figure 11.5.

If there is no edit window on the screen, you will be prompted to create a window. As before, hold the leff mouse button down, move the mouse until it forms a rectangle of an acceptable size and shape, then release the button. Your function definition will automatically appear in this edit window.

!L~nb&A IA Bj (* OJtfJ’ øø:ø1øO:cw ‘~;‘~‘‘ø .~.tr~r
(IF 113’REATEPP A B’i EqV;r~
THEN iOøUUTE "THE ::.pø’.T f ~Ir,GER),l cl,,t’
ELSE 1~UUTE THE .=øE.u’N& j:. eluh’ER;J)) 4ep~:c
/‘‘tC.h
.
Un~io
Find

Rcorint
cit.

EOlfl/C T7~
Sr:ok
Eø.. y

E..t.

Figurø Il.L An Edit Window

Many changes are easily done with the structure editor. Notice that by pressing the left mouse button, different expressions are underlined. Underline BIGGER as in Figure 11.5. Release the left mouse button.

To add an expression that doesn't appear in the edit window, (i.e. it can't simply be underlined), just type it in. Doing this will create an edit buffer below the DEdit window. For example,

type LARGER and hit crø (Remember to cr! You won't be able to do anything in the editor until you cr - this can fool you at first, so beware.) A new window opens up at the bottom for the new expression. (See Figure 11.6.)

LARGER now has the bold line underneath it, while BIGGER has a dotted line.

A

11.4 EDITING ~O ~VING

USING THE LIST STRUCTURE EDITOR

, LAMDOA VA B\ ~ø ødltød 'ø3' øOøc 00 l F;3Q'ø) ArtOr
VV (OREATERP A B) Befom
~ VQUOTE -THE FIRST Is 816OER)) cOIOtO
(15 (QUOTE VTHE SEL'ONO IS BIW~Ry,\.i Ropl&ce
witch
( )

y)out
Unoo
Find
wap

FQum Il.L Edit Window with Edit Buffer

DEdit keeps track of items you have chosen by Using a stack. The underlines tell you the order of the items on the stack. The solid underline indicates the item on the top of the stack; the dotted underline indicates the second to the top. (liIGGER was pushed on first. When LARGER was pushed on, BIGGER became the second element in the "stack", and LARGER the first.)

Many commands operate with two items on the stack. Some of them are listed below:

Atter pops the stack, and adds this top item (in this example, LARGER) to the edit window affer the second item on the stack(in this example, BIGGER). The item that was at the top of the stack, LARGER, will now appear in both the original and the new position.

Before pops the stack, and adds this top item (in this example, LARGER) to the edit window before the second item on the stack. (See Figure 11.7.)

(LAKBDA VA 8' C' oJ'lfG '3,-Oocø~ ~F;l.Oøø ,~rtOr
(IF VGREATERp A 8J E~'inre
~ (QUOTE (THE FIRST IS ~'R ,8øIUGEP); cOlGte

ELI (QUOTE (THE SECOND IS 8I,øb’E .j! ,1J F!Gplace
itch
r

tJut
Undo
Find
,=-.i,1r
P.O~rir,t
Eda

fiUre 11.7. The command Before is chosen; the word LARGER appean
Iefore the word BIGGER

Replace pops the stack, and substitutes this top item for the second item on the stack.

Sat tch changes the position of the first and second items on the stack in the edit window.

Find pops the stack, and searches this top expression for an occurance of the second item on the stack. If the item is found, it is underlined with a solid line, that is, pushed on the stack. To find the next occurance, simply choose "Find" again. If the expression is not found, the prompt window will blink, and a

øspøc1ø11~ asøfa1 If yri ant to &pøcø ~r coøants)
There are other editor commands which can be very UsefUl. To learn about them, read to the lntertis~D Reffrence Manual, Volume 2, Section 16, on DEDIT.

it .4 _ File _ Functions and _ Variables - _ How to _ See Them _ and _ Save Them
With lnterlis~D, all work is done inside the "Lisp Environment". There is no "Operating System" or "Command Level" other than the lnterlis~D Executive Window. All functions and data strUctures are defined and edited using normal Interlisp-D commands. This sertion describes tools in the Interlisp-D environment that will keep track of any changes that you make in the environment that you have not yet saved on files, such as defining new functions, changing the values of variables, or adding new variables. And it then has you save the changes in a file you specify.

11.5 File Variables

Certain system-defined global variables are used by the file package to keep track of the environment as it stands. You can get system information by checking the values of these variables. Two important variables follow.

ø FILELST evaluates to a list, all files that yoU have loaded into the lnteris~D environment.

ø filenameC0liS (Each file loaded into the Lisp environment has associated with it a global variable, whose name is formed by appending "COMS" to the end of the filename.) This variable evaluates to a list of all the functions, variables, bitmaps, windows, and soon, that are stored on that particular file.
For example, if you type:

~FILEC0*s

the system will respond with something like:
FKS YouR-FZRST-Fu*CTIil)
VARS))

11.6 Saving Interlisp-D on Files
The functions (FILES?) and (NAKEFILE ‘filename) are useful when it is time to save function, variables, windows, bitmaps, records and whatever else to files.

message that the item was not found will appear. (See Figure 11.8 for an example of an item, the atom THIRD, not appearing in the function, YOUR-FIRST-FUKCTION.

ø1

L.flFBo~P~ø~T\P _ B! (,‘-.J’l-.J. _ .z’ _ P..,n _ 1-
THEN ‘c1.lcTE _ ‘THE _ FIPT _ ~I.’i’.EP’]
ELSE 1øtlJlJTE _ HE _ ‘/E/l)MlD

TrtI,,,v Sr.i
El

ET.

Figwø 11.& The atom THIRD is not in the fundion being edited
Saap changes places, on the stack, of the first and second items on the stack. The edit window does not change, except that the expression that had a solid underline now has a dotted underline, and vice versa.

Delete works on only the top item of the stack. Delete removes the solid underlined expression from the edit window.
Undo undoes the last editor command.

Completing the example begun earlier, here’s how to have the word LARGER that you typed into the edit buffer appear in place of the BIGGER that you selected from the DEdit window: select the SWITCH command. Notice that the two items are switched, and the stack is popped. Now select EXIT and to leave the editor, and your function will again be redefined.

11.3.1 Commenting Fundions

Tert can be marked as a comment by nesting it in a set of parentheses with a star immediately after the left parenthesis.
(ø This ii thø Von of ø c~rtt)

Inside an editor window, the comment will be printed in a smaller font and may be moved to the far right of the code. Sometimes, however, centered comments are more appropriate. To center a comment, type ,, ... after the left parenthesis.
ø This co.oortt ø111 rtot bø rnd to thø ?ør ri9ht of thø
co5oø but ø111 bø cørttørd)

It is also possible to insert Iinebreaks within a comment. A dash should be placed in the comment whcrevør A carriagø return is needed. Thii feoturø allows several commønt1 to bø placed insidø one S.t of pirøntheses.

(ø This cołoo.t ø111 h t~~ at. tøo 1iøos. -

11.6 FIrING AND LAYING

SAVING INTERLlSP-D ON FILES

(FILES?) displays a list of variables that have values and are not already a part of any file, and then the functions that are not already part of any file.

Type:

(FILES?)

the system will respond with something like:
tbø variables: ~.VARIlLE cURREKT.tuRTLE.. to be du;ed.
thø functions: RI6HT LEFT FOIAff liCK*Aa cLEAR-uREEil.. to be d~~.

srit to s&y øbere thø abovø go?

If you type Y, the system will prompt with each item. There are three options:

(1) To save the item, type the filename (unquoted) of the file where the item should be placed. (This can be a brand new file or an existing file.)

(2) To skip the item, without removing it from consideration the next time (FILES?) is called, type crø This will allow you to postpone the decision about where to save the item.
(3) If the item should not be saved at all, type J. NoilhQ re will appear afler the item.

Part of an example interaction is shown in the following figure:

HIL

u31~(FILES,)

Che variables: MY-'y'AR. To be di.imped.
the functions: MY-SEcuNO-FUtllJTIJN,
YJUP-FIPøoT'FUNi)TIJN
to be dumped.

want to say where the .ibove 30 ø? 'ye'
(variables)
NY-VAR Nowhere
(runctions)

NY-SELrnNO-FUN&'TION File name: E;~AMPL~

F~11.9. Part of an interaction using the function FILES?
(FILES?) assembles the items by adding them to the appropriate file's COMS variable. (See Section 11.5, Page 11.7.)
(FILES?) does NOT write the file to secondary storage (disks or floppies). It only upclates the global variables discussed in SectionIt.S.

(NAKEFILE 'Tl lenaøe) actually writes the file to secondary storage. Files should only be writen when the time is set. If the time is not set, you will run into problems, such as not being able to copy your file. To check

the time, typø
(riTE)

If the date is correct, yoU can safely use IRE FILE. If it is riot correct, set the time with the function SETTIKE. To use it, type (SETTIKE date), where datø isa string such as the one shown inFiguretl.10.

it.a Eomlflll ANC SAVING
I

SAVING INTERUSP~ ON FILES

NIL

97;k(SETTIME "10-Jul-86 15:08 2<8)
"i6-Jul-86 15;08:22 EDT"
98+

Fqøurn 11.10. Using the SETTIKE function to set the date and time
Once the time is set correctly, use the function MAKEFILE. Type:
(liffEFILE ‘P.FILE.~)

and the system will create the file. The function returns the full name of the file created. (i.e. (DSK)MY.FlLE.NAME.; 1).
Note: Files written to (DSK) are permanent files. They can be removed only by the user deleting them or by reformatting the disk.

Other file manipulation functions can be found in Section 8.6, Page 8.3.

----- Next Message -----

Date: 19 Dec 91 15:20 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.152031pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11670>; Thu, 19 Dec 1991 15:20:42 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 15:20:31 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

t3. FLEXIBILITY AND FORGIVENESS:
CLISP AND DWIM

CLlSP, (Conversational Lisp), and DWlM, (Do What Mean), are two Interlisp utilities that make life easier.

13.1 CLlSP

CLlSP allows the machine to understand and execute commands given in a non-standard way. For example, Figure 13.1 contains an example expressi on (4 + 5).

NIL

b’4-iJ + 5;
9

85’

F9ure 13.1. cLlsP allows the use of infix notation

Without CLlSP, you would need to type this using the notation (PLUS 4 5). CLlSP allows you to use expressions such as (4 + 5) for all arithmetic expressions.

CLlSP also allows you to use more readable forms inrtead of standard Lisp control structures. Expressions like IF-THEN-ELSE statements can replace COND statements. For example, instead of:

(CIO 1J6RE(APLTUESRPBA B (PLUS A 10))
10

the following can be used:

(if (A ~ B) then (A + 10) else (B + 10))

The system translates this CLlSP code into Interlisp-D code. Setting flags will allow you to either save the CLlSP code, or save the translation. One such flag is CLISPIFTRANFLG; if it is set to ffIL, all the IF statements will be replaced with the equivilent CORD statements. This means that when you DEdit the function, the IF will be removed and replaced with the CORD. Typically, flags such as this one are set in your INlT file. These flags are dixussed in the Intertlsp-D Reference Manual in Volume 2, Section 21.

OWlM

13.2 DWlM

DWlM tries to match unrecognized variable and function names to known ones. This allows Lisp to interpret minor typing errors or misspellings in a function, without causing a break. Line 87 of Figure 13.2 illustrates how the misspelled 0ANNANNA was replaced by 8ANANA before the expression was evaluated.

NIL

a7(8ETQø 8~øN.HA ‘FRUITj
FRUIT

38’8nNN,,~NNA
=8,,’H,,NA
FRUIT
39’

Figure 13.2. Examples of CLlSP and DWlM features

Sometimes DWlM may alter an expression you didn’t want it to. This may occur if, for example,a hyphenated function name (eg. (NY-FUNCTION)) is misused. If the system doesn’t recognize it, it may think you are trying to subtract "FUN~lON" from "MY". DWlM also takes the liberty of updating the function, so it will have to be fixed. However, this is as much a blessing as a curse, since it points out the misused expression!

13.2 F~1lUM AND ~ROVENES$: cub AND OWN
I

----- End Forwarded Messages -----

Figure 13.2. Examples of CLlSP and DWlM features

Sometimes DWlM may alter an expression you didn’t want it to. This may occur if, for example,a hyphenated function name (eg. (NY-FUNCTION)) is misused. If the system doesn’t recognize it, it may think you are trying to subtract "FUN~lON" from "MY". DWlM also takes the liberty of updating the function, so it will have to be fixed. However, this is as much a blessing as a curse, since it points out the misused expression!

13.2 F~1lUM AND ~ROVENES$: cub AND OWN
I

----- End Forwarded Messages -----

Second Group

Date: 19 Dec 91 18:11 PST (Thursday)
Posted-Date: 19 Dec 91 18:19 PST
From: John Sybalsky:PARC:Xerox
Subject: more primer files.
To: porter:mv:envos

>>CoveringMessage<<

----- Begin Forwarded Messages -----

Date: 19 Dec 91 15:28 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.152817pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11662>; Thu, 19 Dec 1991 15:28:23 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 15:28:17 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

F- 14. BREAKPACliGE

The Break Package is a part of Interlisp that makes debugging your programs much easier.

14.1 Break WindoNT

A break is a function either called by the programmer or by the system when an error has occurred. A separate window opens for each break. This window works much like the Interlisp-D Executive Window, except for extra menus unique to a break window. Inside a break window, you can examine variables, look at the call stack at the time of the break, or call the editor. Each successive break opens a new window, where you can execute functions without disturbing the original system stack. These windows disappear when you resolve the break and return to a higher level.

14.2 Break Package Example

This example illustrates the basic break package functions. A more complete explanation of the breaking functions, and the break package will follow.

The correct definition of FAGTORIAL is:
(DEFIKEQ (FMT0RIAL (xj
then 1

(iføf5ø (ITIES x (f~ToRIAL (sue, xj

To demonstrate the break package, we have edited in an error: DUffKY in the IF statement is an unbound atom, it lacks a value.
(DžFIKEQ (F~T0RIAL (xj
then ~

(if~[~~ (ITIKES x (FACTORIAL ~suei xj

The evaluated function
(F~T0RI~ 4)

should return 24, but the above function has an error. DUMMY is an unbound atom, an atom without an assigned value, so Lisp will "break". A break window appears (Figure 14.1), that has all the functionality of the typing Interlisp-D expressions into the lnterlis~D executive window (The top level), in addition to the break menu functions. Each consecutive break will move to another level "down".

BREAK PACKAGE 141

BREAK PACKAGE EXAMPLE

51+(PP Fllu’T&RIAL)
cFACTORlAL

[LA’NBOR ! ‘.j "łrOMnžNT~ł
(if (EROP ‘~
i,,ien Dummy

6Jil (lTIflEc A !FR~TORIAL !.UB1 :~j;
!FACTCPIALj
5?(FALTORIAL 4,1

DUMMY (in FAi’,TORlALJ in =ERDP P1!t4flY
only br’okøon!

Figuro I..l. Break window

Move the mouse cursor into the break window and hold down the middle mouse button. The Break Menu will appear. Choose BT. Another menu, called the stack menu, will appear beside the break window. Choosing stack items from this menu will display another window. This window displays the function’s local variable bindings, or values. (See Figure 14.2) This new window, titled FACTORlAL Frame, is an inspector window. (See inspector Chapter 32).
Sr

fau’TUR[AL

EP.PoM5ET
fiRE&1

UNBOUND ATOM LflQ
DUMMY (in fAcTORIAL) in \(ZEROP x) DUMMY) cob ø
FLøiDRI~

(DUMMY broken) cob
FkWRl~
L.OB

F,c~RI~
L’4M0

Figun 14.3. Back Yraco of trio 5ystem Stack

From the break window, you can call the editor for the function FACTORIAL by typing
(OF F~15IL)

Underline X. Choose EVAL from the zditor menu. The value of X at the time of thff break will appear in the edit buffer below tho editor window. Any list or atom can be evaluated in this way (See Figure 14.3.)

14.1 lRF~PACMA’GF

BREAK PACKAGE EXAMPLE

UNBOUND ATOM

ø DUMMY (in FACTORIAL ~ (ITIKES x \øfASTORIAL ~SUB1 X))))) Replace
switch

( )

ø (DUMMY broken) ()cUt

OF FAL’TORIAL) Undo

Find
Swop
Reprint
Edit

EatCam
Break
E~a1
E.t

Figure 14.3. Editing from the Break Window

Replace the unbound atom DUffNY with 1 ø Exit the editor with the EXIT command on the editor menu.

The function is fixed, and you can restart it from the last call on the stack (It does not have to be started again from the Top Level) To begin again from the last call on the stack, choose the last (top) FACTOR1AL call in the BT menu. Select REVERT from the middle button break window, or type it into the window. TThe break window will close, and a new one will appear with the message: FACTORlAL broken.

To start execution with this last call to FACTORIAL, choose OK from the middle button break menu. The break window will disappear, and the correct answer, 24, will be returned to the top level.

14.3 _ Ways to _ Stop _ Execution _ from the _ Keyboard, called _ "Breaking _ Lisp"
There are ways you can stop execution from the keyboard. They differ in terms of how much of the current operating state is saved:

Control-G provides you with a menu of processes to Interrupt. Your process will usually be ø‘ EXEC". Choose it to break your process. A break window will then appear.

Control-B causes your function to break, saves the stack, then displays a break window with all the usual break functions.
For information on other interrupt characcers, see the Interlisp Reference Manual, volume 111, page 30.1.

PROGRAMMING BREAKS AND DEBUGGlNG CODE

14.4 Programming Breaks and Debugging Code
PrOgramming breaks are put into code to cause a break when

that section of code is executed. This is very useful for debugging code. There are 2 basic ways to set prOgramming breaks:

(BREAK functionna:e) This function call made at the tOp level will cause a break at the start of the execution of "functionname". This is helpful in checking the values of parameters given to the function.

Setting a break in the editor Take the function that you want tO break into the editor. Underline the expression that should break before it is evaluated. Choose BREAK on the editor command menu. Exit the editor. The function will break at this spot when it is executed.

Once the function is broken, an effective way tO use the break window for debugging is to put it into the editor window. (See Section 14.2, Page 14.2.) All the local bindings still exist, so you can use the editor's EVAL command to evaluate lists, variables, and expressions individually. Just underline the item in the usual way (move the mouse to the word or parenthesis and press the leff mouse button), then choose EVAL from the command menu. (See Section 14.2 for more detail.)

Both kinds of programmed breaks can be undone using the (UNBREAK) function. Type
(~KBRDF functionnm)

Calling (UNBREAK) without specifying a function name will unbreak all broken functions.

14.5 Break Menu

Move the mouse cursor into the break window. Hold the middle button down, and a new menu will pop up, like the one in Figure 14.4.

OK
BT
BY!
"a

f~ure 14.& Thø middle bUtton menu in the Break window

Five of the selection& are particularly important when just starting to use lnterlis~D:

8T Sack Trace displays the stack in a menu beside the break window. Back Trace is a very powerful debugging t00l. Each function call is placed on tho stack and removed when the execution of that function is complete. Choosing an item on thø stack will open another window displaying that item's local

voriobles and their bindings. This is on inspector window thit offers all the power of the inspector. (For details, see the section on the Inspector, Chapter 32).

? Sefore you use this menu option, display the stack by choosing 8T from this menu, and choose a function from it. Now, choose 7: It will display the current values of the arguments to the function that has been chosen from the stack.

~ Move back to the previous break window, or if there is no other break window, back to the top level, the InterlispøD Executive Window.

REVERT Move the point of execution back to a specified function call before the error. The function to revert back to is, by default, the last function call before the break. If, however, a different function call is chosen on the BT menu, revert will go back to the start of this function and open a new break window. The items on the stack above the new starting place will no longer exist. This is used in the tutorial example. (See Section 14.2, Page 14.1.)
OK Continue execution from the point of the break. This is useful if you have a simple error, i.e. an unbound variable or a nonnumeric argument to an arithmetic function. Reset the variable in the break window, then select OK. (See Section 14.2.)
(Note: In addition to being available on the middle button menu of the break window, all of these functions can be typed directly into the window. Only ST behaves differently when typed. It types the stack into the trace window instead of opening a new window.)

14.6 Returning to Top Level

Typing Control-D will immediately take you to the top level from any break window. The functions called before the break will stop, but any side effects of the function that occurred before the break remain. For example, if a function set a global variable before it broke, the variable will still be set afler typing Control-D.

BREAK PACKAGE 14.5
1

----- Next Message -----

Date: 19 Dec 91 15:51 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.155149pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11668>; Thu, 19 Dec 1991 15:51:54 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 15:51:49 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

27. WINDOWS AND REGIONS

27.1 Windows

Windows have two basic parts: an area Ofi the screen containing a collection of pixels, and a property list. The window properties determine how the window looks, the menus that can be accessed from it, what should happen when the mouse is inside the window and a mouse button is pressed, and soon.

27.1.1 CREATEW

5ome of the window’s properties can be specified when a window is created with the function CREATEW. In particular, it is easy to specify the size and position of the window; its title; and the width of its borders.

(CREATEW region title borderw'idth)

Region is a record, named REGION, with the fields left, botto:. width, and height. A region describes a rectangular area on the screen, the window's dimensions and position. The fields left and bottoø refer to the position of the bottom leff corner of the region on the screen. Vi dth and height refer to the width and height of the region. The usable space inside the window will be smaller than the width and height, because some of the window's region is consumed by the title bar, and some is taken by the borders.

Title is a string that will be placed in the title bar of the window. Bordervvidtfr is the width of the border around the exterior of the window, in number of pixels.
For example, typing:
(SETQ ~.WIN~ CREATEW
(CREAT RE6IS loo 150 300 200)
THIS Is ~ r"w ilIN~ø)

produces a window with a default borderwidth. Note that you did not need to specify all the window's properties. (See Figure 27.1.)

ø ,.J[1.lGJ'J (øflf,,Tfff i'øCRE"TEPE,IrnN jvjw 5- '9;, ø"~~i
"TriI;' I> My ij'lN 'ff Ibl&U'M' ii

(~[N&JwlM2.65554

FigUre 27.1. Creating a Window

In fact, if (CREATEW) is called without specifying a region, you will be prompted to sweep out a region for the window. (See Section 10.2, Page 10.2.)

27.1.2 WlNDOWPROP

The function to access or add to any property of a window's property list is WIliDOVPROP.
(WIN~PR0P window property <value>)

When you use WIKDOWPROP with only two arguments - window and property - it returns the value of the window's property.
When you use wIKOOVPROP with all three arguments - window, property and value - it sets the value the window's property to the value you inserted for the third argument.

For example, consider the window, NY WINDOW, created using (CREATEW). TITLE and REGION are both properties. Type
(ilI*~PW :.uI~ 'TITLE)

and the value of MY.WlNDOW's TITLE property is returned, "THIS 15 MY OWN WINDOW". To change the title, use the WINDOWPROP function, and give it the window, the property title, and the new title of the window.
(wIK~PW ~.uI~ 'TITLE øP FIRST ilIK~ø)

automatically changes the title and automatically updates the
window. Now the window looks like Figure 27.2.

27.1 w1N00WS AND REG~NS

WINDOWS

7øt'WINDOWfROP NV WINDOW TITLE)
IS NV OWN WINDOW"

s.(WINDOWPROP NY.WlNDOl4 TITLE 'øQY FIRST WINDOW")
THIS IS M\' OWN WINDOW"
4'.

FigUre 27.2. TITLE is a Window Property

Altering the region of the window, NY. VINDOV, is also be done
with vINDOWPROP, in the same way you changed the title.
(Note: changing either of the first two numbers of a region
changes the position of the window on the screen. Changing
either of the last two numbers changes the dimensions of the
window itself.)

27.1.3 Getting windows to do things
Four basic window properties will be discussed here. They are
CURSORINFN, CURSOROUTFN, CURSORffOVEDFN, and
BUTTONEVENTFN.

A function can be stored as the value of the CURSORlNFN
property of a window. It is called when the mouse cursor is
moved into that window.

Look at the following example:

(1) First, create a window called MY.WlNDOW. Type:
(SETQ P.WINDQW
(CREATEI

(cREATERE6Ia 200 200 200 200)
"THIS WIllDOW WILL IREMlø))

This creates a window.

(2) Now define the function SCREAMER. It will be stored on the
property CURSOR1NFN. (Notice that this function has one
argument, WlNDOWNAME. All functions called from the
property CURSOR1NFN are passed the window it was called from.
So the value of MY. WINDOW is bound to WlNDOWNAME. When
it is called, SCREAMER simply rings bells.
(DEFINžQ (ScREMER (WIK~~E)
RIilBELLS)

PROlPTPRIlT TAT - IT WDRFSI")

RIKBELLS)))

(3) Now, alter that window's CURSORINFN property, so that the
system calls the function SCREAMER at the appropriate time.
Type:

(WIN~PRoP P.wINI;0II ‘cuR~RIafø
(F~IIk:TIK IR~R))

(4) Affer this, when you move the mouse cursor into MY.WlNDOW, the CURSORINFK property’s function is called, and it rings beJls tvvice.

CURSORINFN is one of the many window properties that come with each window - just as REGION and TITLE did. Other properties include:

CURSOROUTFN The function that is the value of this property is executed when the cursor is moved out of a window;

CURSORMOVEDFN the function that is the value of this property is executed when the cursor is moved while it is inside the window;

BUTTONEVENTFN the function that is the value of this property is executed when either the Ieff or middle mouse buttons are pressed (or released).

Figure 27.3 shows MY.WlNDOW’s properties. Notice that the CURSORINFK has the function SCREAMER stored in it. The properties were shown in this window using the function INSPECT. INSPECT is covered in Chapter 32.

. . ‘ 1 ø
GREEN NIL

HI NOo’rtENTR’[FN O liE. TT’(PRObES
PRfllESS NIL
‘,‘181)ROER 4
NEWREL’I)NF4 NIL

‘NTITLE øTHIS ‘ffiINDOW ‘tILL .QCREAn!"
MOlEfN NIL
CLOSEFN NIL
HORIZOCROLL’.yIND1)’,t NIL
"ER1L’ROLLNINoO’ff NIL
c.u’ROLLFN NIL
H)RI=J-’cRlLLREG NIL
":‘ERTSCR)LLREU NIL
USERDATA NIL
E!’!TENT NIL
REOH4PEFN NIL
REPAINTFN NIL
L’URSORttOvEDFN NIL
CURSOROUTFN NIL
CURSORINFN SCCE’øThER
RIGHTBUTTONFN NIL
BU1FONEVENTFN TOTOPU
REG 12J0 "L)9 øJ~ ‘36!
SavE (BITMAP~øł3,1jo52ł
NE~(’t (WIflD1)’-’1j55,1’lj’..ø8
DSP ~5TRE>M\,ø~øF,jjjj~4

Figurø 27.3. Inspeaing MY.wlNDow for MouseRelated Window Properties

You can define functions for the values of the properties CURSOROUTFK and CURSORMOVEDfN in much the same way as you did for CURSORINfN. The function that is the value of the property BUTTOHEVENTFN, however, cab be specialized to respond in different ways, depending on which mouse button is pressed. This is explained in the next section.

27.1.3.1 BUtrONžVENTFN

BUTTONEVENTFK is anothqr property of a window. Tho function that is stored as tflø valuø of this property is called when tho mouso is insidø tho window, and a mouso button is pressed. As an examplø of how to usø iL type:

witurows

(wI~PKP :.ilIK~ ‘euTTW"EKTtr
(F~TI5 ScREAøER))

When the mouse cursor is moved into the window, bells will ring because of the CURS0RlNFN, but it will also ring bells when either the Jeff or middle mouse button is pressed. Notice that the right mouse button functions .5 it usually does, with the window manipulation menu. If only the left button should evoke the function SCREAMER, then the function can be written to do just this, using the function MOUSESTATE, and a form that only NOUSESTATE understands, ONLY. For example:
(DEFIKEQ

(SCRElERZ WIK~)
(if ESTATE (aLY LEFT))
thøa (RIKBžLLS))))

In addition to (ONLY LEFT), MOUSESTATE can also be passed (ONLY MIDDLE), (ONLY RIGHT) or combinations of these (e.g. (OR (ONLY LEFT) (ONLY MIDDLE))). You do not need to use ONLY with MOUSESTATE for every application. ONLY means that that button is pressed and no other.
If you do write a function using (ONLY RIGHT), be sure that your function also checks position of the mouse cursor. Even if you want your function to be executed when the mouse cursor is inside the window and the right button is pressed, there is a convention that the function DOVINDOWCOM should be executed when the mouse cursor is in the title bar or the border of the window and the right mouse button is pressed. Please program your windows using this tradition! For more information, please see the Intertisp-D Reference Manual, Volume 3, Chapter 28, Pages 7 and 28.

Please refer to the Intertisp Reference Manual, Volume 3, Chapter 28, for more detail and other important functions.

27.1.4 Looking at a window’s prOperties

INSPECT is a function that displays a list of the properties of a window, and their values. Figure 27.3 shows the INSPECT function run with MYøWINDOV. Note the properties introduced in CREATEW: WBORDER is the window’s border, REG is the region, and WTITLE is the window’s title.

27.2 Regions

A region is a record, with the fields LEFT, BOTTOM, WIDTH, AND HEIGHT. LEFT and BOflOM refer to where the bottom leff hand corner of the region is positioned on the screen. WIDTH and HEIGHT refer to the width and height of the region.
CREATERE6ION creates an instance of a record of type REGION. Type:

(SETO ~.RE6Ia (CREATERESIl 15 loo 200 450))

REGIONS

to create a record of type REGION that denotes a rectangle 200 pixels high, and 450 pixels wide, whose bottom leff corner is at position (15, 100). This record instance can be passed to any function that requires a region as an argument, such as CREATEV, above.

a,. WlN00WS ANO REGIONS

----- Next Message -----

Date: 19 Dec 91 15:59 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.155935pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11672>; Thu, 19 Dec 1991 15:59:45 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 15:59:35 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

28. WHAT ARE MENUS?

While Interlisp-D provides a number of menus of its own (see Section 7.1, Page 7.2), this section addresses the menus you wish to create. You will learn how to create a menu, display a menu, and define functions that make your menu useful.
Menu's are instances of records (see Chapter 24). There are 27 fields that determine the composition of every menu. Because Interlisp-O provides default values for most of these descriptive fields, you need to familiarize yourself with only a few that we describe in this section.

Two of these fields, the TITLE of your menu, and the ITEMS you wish it to contain, can be typed into the InterlispøD Executive window as shown below:

NIL

33'(.ETO MY. MEN (cRE"'TE ME/lb
TiTLE ,.PLE~~SE CHCio8ž ONE OF THE
ITEMS"

ITEMS (0,LlIT NE,T-l)UE;STION
NE;~T-TOPIL SEE-TOPIC;5'JJJ
,rMENU!,#c4, ij'ø:'3jH

Figure 28.1. Creating a menu

Note that creating a menu does not display it. MY.MENU is set to an instance of a menu record that specifies how the menu will look, but the menu is not displayed.

28.1 Displaying Menus

Typing either the MENU or ADDNENU functions will display your menu on the screen. MENU implements pop-up menus, like the Background Menu or the Window Menu. ADDMEHU puts menus into a semi-permanent window on the screen, and lets you select items from it.

(MENU MENU POSITION) pops-up a menu at a particular position on the screen.
Type:

(*EKU MY.ffI KIL)

to position the menu at the end of the mouse cursor Note that the POSITION argument is NIL. In order to go on, you must either choose an item, or move outside the menu window and

press a mouse button. When you do either, the menu will disappear. If you choose an item, then want to choose another, the menu must be redisplayed.

(ADONENU menu window position) positions a permanent menu on the screen, or ;n an existing window.

Type:

(ADlEKU P.*EI)

to display the menu as shown in Figure 28.2. This menu will remain active, (will stay on the screen) without stopping all the other processes. Because ADONEliU can display a menu without stopping all other processes, it is very popular in users programs.
If window is specified, the menu is displayed in that window. If window is not specified, a window the correct size for the menu is created, and the menu is displayed in that window.
If position is not specified, the menu appears at the current position of the mouse cursor.

NE..TQøUESIlCN

3EEToPIC> .
.

Figure 28.2. A Simple Menu, displayed with AooNriU.

28.2 Getting Menus to DO Stuff
One way to make a menu do things is to specify more about the menu items. Instead of items simply being the strings or atoms that will appear in the menu, items can be lists, each list with three elements. (See Figure 28.3.) The first element of each list is what will appear in the menu; the second expression is what is evaluated, and the results of the evaluation returned, when the item is selected; and the third expression is the expression that should be printed in the Prompt window when a mouse button is held down while the mouse is pointing to that menu item. This third item should be thought of as help text for the user. If the third element of the list is NIL, the system responds with "Will select this item when you release the button".

NIL

17+(SETQ Nv.MENU2 (SRžATE MENU

TITLE "PLEASE LHOOSE ONE OF TflE ITEMS"
I~,EMS ‘(VQUIT
(PRINT "STOPPEO" \
"LHOOSE THIS TO 5O~ø’’,’

(NE\T-QUESTIOH

(PRINT "HERE IS TME NE.’\’T QLlžSTIOH .
øu’HOOSE THIS TO ~E lSKED THE NE."T QUESTION"’,

iNE!~T-TOPIL

(PRINT øøHERE IS THE NE’~T TOPIL .
"C.HOOSE THIS TO KOvž OH TO THE NE’\T SueJELT" ‘1

(SEE-TOPICS

(PRINT "THE FOLLOYIN6 HA’\E NOT e.EEN LžARNEO"’,
*CHOOSE THIS TO SEE THE TOPICS NOT YET LErtRNEO"l ‘ii
ø~~MENU,’#5~. ‘.5~5j
1qL(cl&MENL MY. MEtlU:’
,rNIN&El’~~~4’, 175350
14

Firnre 28.3. Creating a menu that will do things, then displaying it with the funttion ADDNEHU

Now when an item is selected from KY.KENU2, something will happen. When a mouse button is held down, the expression typed as the third element in the item’s specification will be printed in the Prompt window. (See Figure 28.4.)

NE7.T.’JUE’=TlE’r~J
SEE-TOPIC’

Fiqrnre 28.1. Mouse Button Held Down While Mouse Cursor SeIe~ NEXT-QUESTIoN

When the mouse button is released (i.e. the item is selected) the expression that was typed as the second element of the item’s specification will be run. (See Figure 28.5.)

Y-’OUE’Tl"N
‘EETOPlr"’
"HERE IS THE NEXT ilUETION.
Figure 28.5. NEXT-QUESTION Selected

GEHlNG MENUS TO DO STUFF

28.2.1 _ The WHENHžLDFN _ and WNENSELžCTEDFN fields of a _ menu
Another way to get a menu to do things is to define functions, and make them the values of the menu’s WHENHELDFN and WHENSELECTEDFN fields. As the value of the WHENHELDFN field of a menu, the function you defined will be executed when you press and hold a mouse button inside the menu. As the value of the WHENSžLžCTEDFN field of a menu, the function you defined will be executed when you choose a menu item. This example has the same functionality as the previous example, where each menu item was entered as a list of three items.

As an example, type in these two functions so that they can be
executed when the menu is created and displayed:
(DEFIKEQ LžCTED

(SELEcfQPiNTEENNUJSENHENHELO (ITEl.SžLECTED a:. FROM BUTT:. PRESSED)
QUIT (PROMPTPRIKT øcHOOSE THIS TO sToPø))

NEXT-QUESTION (PROMPTPRIKT CHOOSE THIS TO BE ASKED TNE NEXT QUESTION-))
NEXT-TOPIC PROMPTPRINT øCHOOSE THIS TO MOO,E a TO THE NEXT SUBUIECTø))
SEE-TOPICS PROMPTPRINT øCHDOSE THIS TO SEE THE TOPICS NOT YET LžARNEDø))
ERROR (PROM TPRIKT NO liTCH FOUNDø)))))

(DEFINEQ WENSELECTED (ITEM.SELECTED MENU. FROM 8UTT:.PRESSED)
QUIT (PRINT øSTOPPEDø))

NEXT-QU RINT "HERE IS THE NEXT QUESTION...))
NEXT-T øHERE IS THE NEXT TOPIC. .
- PICS PRINT øTHE FOLLONIK HAVE NOT 8EEN LEARNED. ..ø

ERROR (PRONFTPRINT NO liTCH FOUND)))))

Now, to create the menu, type:
(SETQ MY.NE:3 (CREATE NE:
TITLE øPLEASE CHOOSE :E OF THE ITEMSø
ITEK ‘(QUIT NEXT-QUESTION NEXT-TOPIC SEE-TOPICS)
NHENHELDFN (FUNCTIK MY.NENU3.NHENHELD)
fflENSELECTEDFN (FUNCTION NY, .MENU3 .fflENSELECTED)))

Type

(ADDMENU MY.MENU3)
to see your menu work.

NOW, due to executing the WHžNNELDFN function, holding
down any mouse button while pointing to a menu item will
display an explanation of the item in the prompt window. The
screen will once again look like Figure 28.4 when the mouse
button is held when the mouse cursor is pointing to the item
NEXT-TOPIC.

Now due to executing the WHENSELECTEDFN function, releasing
the mouse button to select an item will cause the proper actions
for that item to be taken. The screen will once again look like
Figure 28.5 when the item NEXT-TOPIC is selected.
The crucial thing to note is that the functions you defined for
WHENHELDFN and WHENSELECTEDFN are automatically given
the following arguments:

(t) the item that was sølected, ITEM. SELECTED;
(2) the menu it was selected from, MENU. FROM;
(3) and the mousø button that was pressed BUTTON PRESSED.
Hotø: thesø functions, *Y.NENU3.fflENflELO and
ffY.KEKUJ.ilHEKSELCTEO, wøre quoted using FUKCTIOK
instead of QUOTE both for program røadability and so that the

compiler con produce foster code when the program is compiled.
It is good style to quote functions in Intertisp by using the
function FUNCTION instead of QUOTE.

28.3 Looking at a menuøs fields
INSPECT is a function that displays a list of the fields of a menu, and their values. The Figure 28.6 shows the various fields of NY .NENU3 when the function (INSPECT NY øNENU) was called. Notice the values that were assigned by the examples, and all the defaults.

\JN"PELT NY liENl./3øøl
øl1IHDU'wJ#'1, 54øscj

NENllPECICNB1:TTi=fl o

Imrni';E (ø!VINDLlrtø$#b1.lł5lSjl

ø t1)UlT HEø~T-LløL'E'TI1=1N 'ø'-Ti'iFl'ø ET
ø MENUPOffoø'

ø ANUEAFF'ETFLL: NIL

ffENUEQHT i:FclNTPc:::cf IpTclFt -a
TITLE 'øPLEAL'E CHil.l 'HE ,iF THE ITE
ø ffEHlJoFF6ET A
LECTEDFN fly flEflJ,ø h.ørtEf:EL.FCTEžl

'1flE'flELDFH NV flEPlLl3 \ørtEHHELJP
ø ENl)NHELoFH l:LFF'RCHPT

ø flENOFEEOe4l,'r.FLG NIL
Figure 28.6. The Fields of MY.MENU3

----- Next Message -----

Date: 19 Dec 91 16:10 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.161052pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11680>; Thu, 19 Dec 1991 16:10:56 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:10:52 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

29. lilTMAPS

A bitmap is a retangular array of dots. The dots are called pixels (for picture elements). Each dot, or pixel, is represented by a single bit. When a pixel or bit is turned on (i.e. that bit set to 1), a black dot is inserted into a bitmap. If you have a bitmap of a floppy on your screen, (Figure Figure 29.1), then all of the bits in the area that make up the floppy are turned on, and the surrounding bits are turned off.

FLOPPY
(Ia b~JwP-
~',5,,Bh

(t-:)o

Figure 29.1. Bitmap of a Floppy

BITNAPCREATE creates a bitmap, even though it can’t be seen.
(BIfflPCRDTE width height)

If the width and height are not supplied, the system will prompt
you for them.

EDZTBN edits the bitmap. The syntax of the function is:
(EDITl bitmapname)

Try the following to produce the results in Figure 29.4:
l~SoETiQr:~røB!sTHituPbB~I~PcRDTE eo 40))

To draw In the bitmap, move the mouse into the gridded section of the
bitmap editor, and press and hold the leff mouse button. Move
the mouse around to turn on the bits represented by the spaces
in the grid. Notice that each space in the grid represents one
pixel on the bitmap

To erase Move the mouse into the gridded section of the bitmap editor,
and press and hold the center mouse button. Move the mouse
around to turn off the bits represented by the spaces in the
gridded section of the bitmap editor.

To work on a different section Point with the mouse cursor to the picture of the actual bitmap
(the upper left corner of the bitmap editor). Press and hold the

Jeff mouse button. A menu with the singJe item, ttove will
appear. (See Figure 29.2.) Choose this item.

. .

Figure 29.2. Move the mou5e cursor to the Dtcture of the bitmap. Press and hold
the Iek mouse button. and the Move menu will appear

You will be asked to position a ghost window over the bitmap.
This ghost window represents the portion of the bitmap that you
are currently editing. Place it over the section of the bitmap that
you wish to edit. (See Figure 29.3.)

. .
.

. .

. ... .
.... I .

29.3. .. J=.. :. II.lI:.:;;. _ . .
f1ure Affer you choose move. yoU will be asked to position a ghost

window like this one. Position it by clicking the leff mouse button when the

ghost window is over the part of the picture of the bitmap you would like to edit.
To end the session 8ring the mouse cursor into the upper-right portion of the
window (the grey area) and press the center button. Select OK
from the menu to save your artwork.

29) .lY~

r'. alTMAps

.:: 5''iSETQ ffy IllNAP (I[TNAPcPEATE OR GO)'
j:y.IlfM&P osøt\
,A.BlTMAPlø6',1.q;łlO
58oi,EOIlBM my.IlTNAP\

- -
.

.
-A

fr.j:
'

= "''~
=
. ~.
.

. .

F~ure 29.4. Editing a Bitmap

BITBLT is the primitive function for moving bits (or pixels) from one bitmap to another. It extracts bits from the source bitmap, and combines them in appropriate ways with those of the destination bitmap. The syntax of the function is:

(BITBLT sourcebitmap sourcelefl sourcebottom
destinationbitmap destinationleft destinationbottom width
height sourcetype operation texture clippIngregion)

Here's how it's done - using MY.BlTMAP as the sourcebitmap and MY.WlNDOW as the destinationbitmap.'
(BITBLT rn.BITll4P NIL NIL

P.wIN~ NIL NIL KIL NIL 'INPUT 'REPuCE)

Note that the destination bitmap can be, and usually is, a window. Actually, it is the bitmap of a window, but the system handles that detail for you. Because of the IlLs (meaning "use the default"), MY.BlTMAP will be BlTBLT'd into the lower right hand corner of MY.WlNDOW. (See Figure 29.5.)

~17MAP5

98'(BITBLT KY Strap NIL NIL my ,1(10p,, FL 'IL NIL HIL Tipil' P.žPLlfi

(~=l',

Figure 29.5. 9ITBLTng a Bitmap onto a Window

Here is what each of the SlTBLT arguments to the function mean:

sourcebitmap the bitmap to be moved into the destinationbitmap
sourcelett a number, starting at O for the Jeff edge of the sourcebitmap, that tells SITBLT where to start moving pixels from the sourcebitmap. For example, if the leftmost 10 pixeis of

sourcebitmap were not to be moved, sourceleft should de 10
The default value is O.

sourcebottom a number, starting at O for the bottom edge of the sourcebitmap, that tells BIT6LT where to start moving p1'xels from the sourcebitmap. For example, if the bottom 10 rows of pixels of sourcebitmap were not to be moved, sourcebottom should be 10 The default value is O.

destinationbitmap the bitmap that will receive the sourcebitmap. This is offen a window (actually the bitmap of a window, but Interlisp-b takes care of that for you).

destinationleff a number, starting at O for the leff edge of the destinationbitmap, that tells BITBLT where to start placing pixels from the sourcebitmap. For example, to place the sourcebitmap 10 pixels in from the Jeff, destinationleft should be 10. The default value is 0.

destinationbottom a number, starting at 0 for the bottom edge of the destinationbitmap, that tells BITBLT where to start placing pixels from the sourcebitmap. For example, to place the sourcebitmap 10 pixels up from the bottom, destinationbottom should be 10. The default value is 0.

width how many pixels in each row of sourcebitmap should be moved. The samc amount of space is used in destinationbitmap to receive the sourcebitmap. If this argument is NIL, it defaults to the number of pixels from sourceleft to the end of the row of sourcebitmap.

height how many rows of pixels of sourcebitmap should be moved. The same amount of space is used in destinationbitmap to receive thq sourtebitmap. If this argument is NIL, it defaults to the number of row; from sourcebottom to tho top of the sourcebitmap.

sourcetypø rofors to onø of throø ways to cofivørt thø sourcebitmap for writing. For now, just usø 'INPUT.

29.ø o~ps

' ' ' ' ' ' ' ' ' ' ' ' ' ' '' ' '" ' ' ' ' ' ' ' ' ' ' ' ' '

OIlMAPS

operation refers to how the sourtebitmap gets BlTBLT'd on to the destinationbitmap. 'REPLACE will BLT the exact sourcebitmap. Other operations allow you to AND, OR or XOR the bits from the sourcebitmap onto the bits on the destinationbitmap.
texture Just use NIL for now.
clippingregion just use NIL for now.

Por more information on these operations, see the Interlisp-D Reference Manual, Volume 3, Chapter 27, Page 14.

Sourcebitmap, sourceleft, sourcebottom, destinationbitmap, destinationleft, destinationbottom, width and height are shown in Figure 29.6.

Destination Bitmap
Source Bitmap
FLOPPY
tlcblffkUP'

3/S/Bh height

e./,o

width

Source leh. Source bottom. The "x y coordinates in terms of the source (OOforthewhoiesource).

Destination Jeff, Dertination Bottom. The ,,x y" coordinates in terms of the destination bitmap. (00 to put the source bitmap in the Ieft bottom corner of the dertination bitmap).

Figure 29.6. BITBLT'ed Bitmap of a Floppy

----- Next Message -----

Date: 19 Dec 91 16:16 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.161653pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11679>; Thu, 19 Dec 1991 16:16:57 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:16:53 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

30. DlSPLAYSTREAMS

A displaystream is a generaJized "place to display". They determine exactly what is displayed where. One example of a displaystream is a window. Windows are the only displaystreams that will be used in this chapter. If you want to draw on a bitmap that is not a window, other than with BITBLT, or want to use other types of displaystreams, please refer to the Interlisp-D Reference Manual, Volume 3, Chapter 27.

This chapter explains functions for drawing on displaystreams: DRAWLINE, DRAWTO, DRAVCIRCLE., and FILLCIRCLE. In addition, functions for locating and changIng your curreAt position in the displaystream are covered: DSPXPOSITIOH, DSPYPOSITION, and NOVETO.

30.t Drawing on a Displaystream

Examples will show you how the functions for drawing on a display stream work. First, create a window. Windows are displaystreams, and the one you create will be used for the examples in this chapter. Type:
(SETO EwPLE.wIN~ (CREATEI))

30.1.1 DRAWLlNE

DRAWL IRE draws a line in a displaystream. For example, type:
(DliVLIKE 10 IS loo 150 S øIlERT ExMPLEwIN~)
The results should look like this:

Figure 30.1. The line drawn onto the displayrtream, ExAMPLEwlNDoW

DlSPLAYSTREAMS 30

DRAWING ON A DlSPLAYsTaE:M
The syntax of DRAWL1NE is

(Dli~IKE xl yl x2 y2 width opera tion stream ø)
The coordinates of the Jeff bottom corner of the displaystream
areOO.

xl and yl are the x and y coordinates of the beginning of the line;
x2andy2 are the ending coordinates of the line;
width isthe width of the line, in pixels

operation is the way the line is to be drawn. INVERT causes the line to invert the bits that are already in the displaystream. Drawing a line the second time using INVERT erases the line. For other operations, see the Interlis~D Reference Manual, Volume 111, Page 27.15.

stream is the displaystream. In this case, you used a window.

30.1.2 ORA~O

DRAWTO draws a line that begins at your current position in the displaystream. For example, type:
(Dli~O 120 135 5 ‘IrvERT E~LE.*IH~)
The results should look like this:

Figuro 30.2. Another line drawn onto the displaystream, ExAMPLEøWlNDowø
The syntax of ORAWTO is

(oliilT0 x y width operation stream i)

The line begins at the current position in the displaystream.
x is the x coordinate of the end of the line;
y is they coordinate of the end of the line;
width is the width of the line

operation is the way the lino is to be drawn. INVERT causes the line to invert the bits that aro already in tho displaystream. Drawing a line the second time using INVERT erases the line. For other operations, see the lnteHi~O Reference Manual, Volume Ill, Page 27.15.

stream is the displaystreom. In this case. you used a window.

30.2 llPLAYSTQCANT

DRAW1NG ON A D15PLAr5~E~
30.1.3 DRAWClRCLE

DRAWCIRCLE draws a circle on a displaystream. To use it, type:
(0li~I~LE 150 100 so ‘(~RTICAL 5) KIL E~LE .VI~)
Now your window, EXAMPLE.WlNDOW, should look like this:

Flurø 30.3. The circle drawn onto the displaystream. EXAMPLE WINDOW
The syntax of DRAWCIRCLE is

(0li~IELž centerx centery radius brush dashing stream)
centerx is the x coordinate of the center of the circle
centery is they coordinate of the center of the circle
radius is the radius of the circle in pixels

brush is a list.- The first- item of the list is the shape of the brush. Some of your options include ROUND, SQUARE, and VERTICAL. The second item of that list is the width ofthe brush in pixels.
dashing is a list of positive integers. The brush is "on" for the number of units indicated by the first element of the list, "off" for the number of units indicated by the second element of the list. The third element specifies how long it will be on again, and so forth. The sequence is repeated until the circle has been drawn.
stream is the displaystream. In this case, you used a window.

30.1.3.1 FlLLClRCLE

FILLCIRCLE draws a filled circle on a displaystream. To use it, type:

(FILLCIRCLE 200 150 10 6liY~DE ExlPLE.wIli~)
EXAMPLE.WlNDOW now looks like this:

Figure JO.t A filled circle drawn onto the displaystream, EXAMPLE WINDOW
The syntax of FILLCIRCLE i5

(FILLCIRCLž centerx centery radius texture stream)
centerx is the x coordinate of the center of the circle
centery is theycoordinate of the center of the ci rcle
radius is the radius of the circle in pixels

texture is the shade that will be used to fill in the circle. Interlisp-D provides you with three shades, WHlTESHADE, BLACKSHADE, and GRAYSHADE. You can also create your own shades. For more information on how to do this, see the Interlisp-D Reference Manual, Volumelll, Page 27.7.

stream is the displaystream. In this case, you used a window.
There are many other functions for drawing on a displaystream. Please refer to the Intertisp-D Reference Manual, Volume 111, Chapter 27.

Text can also be placed into displaystreams. To do this, use printing functions such as PRIffl and PRIN2, but supply the name of the displaystream as the "file" to print to. To place the ten in the proper position in the displaystream, see 5ection 30.2, Page 30.4.

30.2 _ Locating _ and _ Changing _ Your _ Position _ in _ a _ Displaystream
There are functions provided to locate, and to change your current position in a displayitream. This can help you place text, and other images where you want them in a displaystream. This primer will only discuss three of these. There are others, and they can be found in the lnterlis~D Reference Manual, Volume Ill, Chapter 27.

30.2.1 DSPXPOSlTlON

DSPXPOSITION is a functiOn that will either change the current x pOsition in a displaystream, or simply report it. To have the function report the current x position in EXAMPLE.WlNDOW, type:

(OSP*PoSlTIoN NIL EXlPLE .ilINDON)

DSPXPOSITION expects two arguments. The first is the new x position. If this argument is NIL, the current position is not changed, merely reported. The second argument is the displaystream.

30.2.2 DSPYPOSlTlON

DSPYPOSITION is an analogous function, but It changes or reports the current y position in a displaystream. As with DSPXPOSlTlON, If the first argument Is a number, the current y position will be changed to that position. If it is NIL, the current position is simply reported. To have the function report the current y position in EXAMPLE.WlNDOW, type:

(DSPYROSITIoN NIL ExlPLE.WIK-~)

30.2.3 MOVETO

The function NOVETO always changes your position in the displaystream. It expects three arguments:
(~-ET0 xystream)

x is the new x position in the display stream
y is the new y position in the display stream

stream is the display stream. The examples so far have used a window.

----- Next Message -----

Date: 19 Dec 91 16:30 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.163054pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11682>; Thu, 19 Dec 1991 16:30:58 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:30:54 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

31. FONTS

This chapter explains fonts and fontdescriptors, what they are and how to use them, so that you can use functions requiring fontdescriptors

You have already been exposed to many fonts in Interlisp-D. For example, when you use the structure editor, DEdit, (See Section 11.3.), you noticed that the comments were printed in a smaller font than the code, and :hat CLlSP words (See Section 13.1, Page 13.1.) were printed in a darker font than the other words in the function. These are only -me of the fonts that are available in Interlisp-D.

In addition to the fonts that appear on your screen, Interlisp-D uses fonts for printers that are different than the ones used for the screen. The fonts used to print to the screen are called DlSPLAYFONTS. The fonts used for prining are called INTERPRESSFONTS, or PRESSFONTS, depending on the type of printer.

31.1 What makes up a FONT?

Fonts are described by family, weight, slope, width, and size. This section discusses each of these, and describes how they affect the font you see on the screen.

Family is one way that fonts can differ. Here are some examples of how "family" affects the look of a font:

CLASSIC This family makes the word "Able" look like this: Able
MODERN This family makes the word "Able" look like this: Able
TERMINAL This family makes the word "Able" look like this: Able
Weight also determines the look of a font. Once again, "Able" will be used as an example, this time only with the Classic family.
A font’s weight can be:
BOLD and look like this: Able
MEDIUM or REGULAR and look like this: Able
The slope of a font is italic or regular. Using the Classic family font again, in a regular weight, the slope affects the font like this:

ITALIC looks like this: A file
REGULAR looks like this: Able

The width of a font is called its "expansion". It can be COMPRESSED, REGULAR, or EXPANDED.

Together, the weight, slope, and expansion of a font specifies the font’s "face". Specifically, the face of a font is a three element list:

(weight slope expansion)

To make it easier to type, when a function requires a font face as an argument, it can be abbreviated with a three character atom. The first specifies the weight, the second the slope, and the third character the expansion. For example, some common font faces are abbreviated:

MRR This is the usual face, MEDIUM, REGULAR, REGULAR;
MlR makes an italic font. It stands for: MEDIUM, ITALIC, REGULAR;
BRR makes a bold font. The abbreviation means: BOLD, REGULAR, REGULAR;

BIR means that the font should be both bold and italic. BIR stands for BOLD, ITALIC, REGULAR.

The above examples are used so oflen, that there are also more mnemonic abbreviations for them. They can also be used to specify a font face for a function that requires a face as an argument. They are:

STANDARD This is the usual face: MEDIUM, REGULAR, REGULAR. It was abbreviated above, MRR;

ITALIC This was abbreviated above as MR, and specifies an italic font;
BOLD of course, makes a bold font. It was abbreviated above, BRR;
BOLDlTALIC means that the font should be both bold and italic: BOLD, ITALIC, REGULAR. It was abbreviated above, BlR.
A font also has a size. It is a positive integer that specifies the height of the font in printers points. A point is, on an 1108 screen, about 1/72 of an inch. On the screen of an 1186, a point is 1/80 of an inch. The size of the font used in this chapter is 10. For comparison, here is an example of a TERMINAL, MRR, size 12 font: Able.

31.2 Fontdescriptors, and FONTCREATE
For InterlispøD to use a fort, it must have a fontdescriptor. A fontdescriptor is a data type in InterlispøD that that holds all the information needed in order to use a particular font. When you print out a fontdescriptor, it looks like this:
[fKTDEIRIPToRjøiø,øs~ø0

Fontdescriptors are created by the function F0NTCREATE. For example,

(F~TCREATE ‘flEL~1lCA 12 ‘~o)

J:

31.2 FOflff

FONTDESCRlPTORS, AND F0NTCREAlE

creates G fontdescriptor that, when used by other functions, prints in HELVETIEA BOLD size 12. Interlisp-D functions that work with fonts Gxpect a fontdescriptor produced with the FONTCREATE function.
The syntax of FONTCREATE is:
(F0KTCREATE family size face)

Remember from the previous section, face is either a three element list, (weight slope expansion), a three character atom abbreviation, e.g. MRR, or one of the mnemonic abbreviations, e.g. STANDARD.

If FONTCREATE is asked to create a fontdescriptor that aJready exists, the existing fontdescriptor is simply returned.

31.3 Display Fonts - Their files, and how to find them
Display fonts require files that contain the bitmaps used to print each character on the screen. All of these files have the extension .DlSPLAYFONT. The file name itself describes the font style and size that uses its bitmaps. For example:
~ERK12.DISPUYFRT

contains bitmaps for the font family MODERN in size 12 points.
Initially, these files are on floppies. The files that are used most offen should be copied onto a directory of your hard disk or fileserver. Usually, this directory is called FONTS.
Wherever you put your .DISPLAYFONT files, you should make this one of the values of the variable DISPLAYFONTDIRECTORIES.
Its value is a list of directories to search for the bitmap files for

display fonts. Usually, it contains the "FONT" directory where you copied the bitmap files, the device (FLOPPY), and the current connected directory The current connected directory is specified by the atom NIL. Here is an example value of DISPLAYFONTDIRECTORIES:

. - 11
NIL

r~':PI:='pL"'yFnNTDIP,ECTBP,IES

i;!Iøo= ' . =PFIL -FnNT~." (D.~fr):!.LIT'.PFIL
fFLnPF"')- NIL!i
9!ø

Figure 31.1. A valueøfor the atom DISFLAYFONTDIRECTORIES When looking for a .DiSPl.AYFONl file. 'he system will check the F0NT directory on the hard disk. then the top level directory on the hard disk, then the floppy. then the current connected dir8rtory

31.4 _ Interpress _ Fonts _ - _ Their files, _ and _ how _ to _ find _ them
Interpress i5 the format that is used by Xerox laser printers. These printers normally have a resolution that is much higher than that of the screen: 300 points per inch.

In order to format f,Ies appropriately for Output on such a printer, Interlisp must know the actual size for each character that is to be printed. This is done through the use of width files that contain font width information for fonts in Interpress format. Initially, these files (with extension .WD) are on floppies. The files should be copied onto a directory of your hard disk or fileserver.

For Interpress fonts, you should make the location of these files one of the 'values of the variable INTERPRESSFOliToIRFcTORIES. Its value is a list of directories to search for the font viidths files for Interpress fonts. Here is an example value of INTERPRE5SFONTD1RECT0R1ES:
. 11
1'lIL

i?IbdTEFPfiETø=:FnN7PIP:EcTnRIž~,~
.i=~~.~
j:~,~

Figure 31.2. A value for the atom INTERPREssFoNTDIREcToRIEs
When looking for a font widths file for an Interpress font, Interlisp-D will cne~ the hard disk.

31.5 Functions for Using Fonts

31.5.1 F0NTPR0P Looking at Font Properties
It is possible to see the properties of a fontdescriptor. This s done with the function FONTPROP. For the following examples, the fontdescriptor used will be the one returned by the function (DEFAULTFONT 'DISPLAY). In other words, the fontdescriptor examined will be the default display font for the system.

There are many properties of a font that might be useful for you.

Some of these are:

FAffILY To see the family of a font descriptor, type:
(FKTPliP (DEFAllLTFoIT ‘DISPLAY) ‘f~ILY)

SIZE As above, this is a positive integer that determines the height of the font in printer’s points. As an example, the SIZE of the current default font is:

31ø ~n

FUNCTIONS FOR USING FONTS

. 11
NIL

Gi,ø(FnNTPROP (DEF~ULTFONT PI~~PLAY)
‘.,ø,:‘IZE\
is,

Figure 31.3. The value of (he font property SIZE of the default font
ASCENT The value of this property is a positive integer, the maximum height of any character in the specified font from the baseline (bottom). The top of the tallest character in the font, then, will be at (BASELINE # ASCE[VT - l). For example, the ASCENT of the default font is:

ø 1 11
NIL

Aø 4’ ,. I!øFnNTPROP if OfF"’ ULTFnNT øPI~,~PL~","!’
‘~e-rENT:!
q.-

A,5~:

Figure 31.& The value of the font property ASCENT of the default font
DESCENT The DESCENT is an integer that specifies the maximum number of points that a character in the font descends below the baseline (e.g. letters such as "p" and "g" have tails that descend below the baseline.). The bottom of the lowest character in the font will be at (BASELINE - DESCENT). To see the DESCENT of the default font, type:

(FOkTPROP (DEfAULTFKT ‘DISPUY) ‘DESr:žKT)
HEIGHT HE IGHT is equal to t’DESCENT-ASCENT).
FACE The value of this property is a list of the form, (weight slope expansion). These are the weight, slope, and expansion described above. You can see each one separately, also. Use the property that you are interested in, VEIGHT, SLOPE, or EXPANSION, instead of FACE as the second argument to FONTPROP.

For other font properties, see the Interlisp-D Reference Manual, VolumeIll, Pages 27.27 - 27.28.

31.5.2 5TRlNGWlDTH

It is offen useful to see how much space is required to print an expression in a particular font. The function STRINGVIDTH does this. For example, type:

(STRIKWIDTH "NV thera!ø (‘L’NTcREATž ‘UCli 10 ‘STAKDARD))
The number returned IS how many leff to right pixels would be

needed if the string were printed in this font. (Note that this

F0NTS 31 S

FUNCTIONS FOR USING FONTS

doesn’t ju5t work for pixels on the screen, but for all kinds of streams. For more information about streams, see Chapter 30.) Compare the number returned from the example call with the number returned when you change GACHA to TlMESROMAN.

31.5.3 DSPFONT - Changing the Font in One Window
The function DSFF0NT changes the font in a single window. As an example of its use, first create a window to write in. Type:
(SETQ ~.FoNT.WINnaN (CttEATE*))

in the Interlisp-D Executive window. Sweep out the window. To print something in the defau!t font, type:
(PRINT ‘HELLO N’f’.FO*T.wIN~)

in the Interlisp-D Executive window. Your window, MY. FONT.WlNDOW, will lOOk sOmething like this:

HELL

Figure 31.5. HELLO, printed with the default font in MY.FONT.WINOOW
Now change the font in the window. Type:
(DSPF0NT (FONTCREATE ‘HELVETICA 12 ‘SOLD) *T.FONT.WINDaN)
in the Interlisp-D Executive window. The arguments to FONTCREATE can be chang~-’d to create any desired font. Now retype the PRINT statement, and your window will look somethinglikethis:

- .
HIL

.q.’~;, PSPFnNT (FnNTrRE~TE ‘HEL";’ET1L~
1:’ø øBnLPt

M’tø.FnNT.vINPnWj

l:FnNTPE~1’RIpTnfl~#?.~,. 1-’ø 14 "4
3~~iPR[NT ‘HELLO MY.fnflr.l]INoniff)
HELLO

Flgurø 31.L The font iiiMY FONT WINDow, changed
Notice the font has been changedl

J.

31.6 FONtt

FUNfll0NS FOR USING F0Nff
31.5.4 _ Globally Changing _ Fonts ________________________
There is a library package to globally change the fonts in all the windows. To use it, first load BlG.DCOM. (See Section 8.6, Page 8.4 for how to load a file.)

To change fonts in 311 windows using the package BlG.DCOM, type

(KE*Fo*T <ke~o~>~

There are four keywords for size of fonts to specify. They are

HUGE, BIG, STANDARD, and MEDIUM. For example:
(*E*FKT ‘BIG)

sets the fonts in ALL the windows to be a larger size. Note: this package changes the fonts everywhere, including the editor window and system merius It is particularly useful to change the size of the font for demos.

31.5.5 Personalizing Your Font Profile
Interlisp-D keeps a list of default font specifications. This list is used to set the font in all windows where the font is not specifically set by the user (Section 31.5.3). The value of the atom FONTPROFILE is this list. (See Figure 31.7.)

A FONTPROFILE is a list of font descriptions that certain system functions access when printing output. It contains specifications for big fonts (used when pretty printing a function to type the function name), small fonts (used for printing comments in the editor), and various other fonts.

F0NTS 317
I

FUNCTIONS FOR USING FONTS

- . . .
43-FJtlTPRUF[LE

l!’ PžF"ULTFClFlT i ,.‘ ‘,‘cH4 LLT;
;ø‘,‘~LHk aj
t’TEPMINk’L Sij

‘.‘BlLPF’lNT :‘ (HžLlETIl=’n’ Jo E,PP.;1
‘lHžL’:’žTIC" L=’ BPP,i
‘llJPEPtl in’ ~FF)

‘,‘LITTLžFC’NT 3 ;ttEL’ø?ErIC" ,3,’
iHE - c 1,1p;

i.’BIC-FCNT ~ llnof - hlIP"i
‘HE 1.=’ BpP.i
‘.‘HEL’ø’ET .- it’ epp:’
‘.‘IrtoPEPrl -
(J\EPFONT 6oLOFElNT
(C.lMllENTFANT LITTLžFi)r’T
‘.‘L"M0P~"FL1flT 61 eFol~! T
i.’.=’r’3TEMFeNT’,i

‘.‘CLI~T’~PFUNT BC’LOF’)1’lf
i.’ CH,,N’3’EF’)HT

i,’PPETT\’1?.Cit1F(ji~T Bl.lLC’FllltlT
i.’FCPlTL DEf"’ULTFEitiT"
,‘Fel’JT"ø 6cLDFclllT;
t.’FCt’1T3 LITTLEFcifølT;
‘.f1tlTJ BlCF(.’r’lT’,i

i.’FElNT~ S ‘,‘HEL’,"ETl;,,’ 10 81P’.
‘lHEL’y’ETlc,, ‘3 61A)
CfillDEPN a 81P:;

t.’FilNTB 6 ‘HEL";’ET16~ 10 8RP’,i
‘.‘HEL’."ET1C~ L’~ BAA"’
llPEPN 3 BAA]

Fi)NT7 ? c"'i'.H~ 1:ø''
:e-"Ln~ 1:ø!'

'.TERMItl,'L 1;''.,,!,
5a,

Figure 31.7. The value of the atom FONTPROFlLE

The list is in the form of an a5sociation list. The font class names, (e.g. DEFAULTFONT, Or SOLDFONT) are the keywords of the association list. When a number follows the keyword, it is the font number for that font class.

The lists following the font class name or number are the font specifications, in a form that the function FONTCREATE can use. The first font specification list affer a keyword is the specification for printing to windows. The list, (GACHA 10), in the figure above is an example of the default specification for the printing to windows. The last two font specification lists are for Press and Interpress file printing, respectively. For more information, see the lnterlis~D Reference Manual, Volume 3, Chapter 27.

Now, to change your default font settings, change the value of the variable FONTPROFILž. lnterlis~D has a list of profiles stored as the value of the atom FONTDEFS. Choose the profile to use, then install it as the default FONTPROFILE.
Evaluate the atom FONTDEFS and notice that each profile list begins with a keyword. (5ee Figure 31.8.) This keyword corresponds to the size of the fonts included. BIG, SMALL, and STANDARD are some of :he keywords foT profiles on this list - SMALL and STANDARD appear in Figure 31.8.

31.8 F0Htt
1

FUNCTl0NS F0R USING F0NTS
[[SMALL cFONTPRQFlLE
(DEFALlLTFONT l (TERMINAL
8)

tøUaCHA 8)
'TERmIHAL 8))

(8OLPFL~NT (Mi!OERtt 3 BRR)
\HELY'FTIL" 6 BRR)
ltl\flEfiH 8 BRfi))
1 LITTLEFCNT ~'ø
(hllCiERN 8 MIR)
lHEL'v'ETIu'"' 8 MIR)
iMCiPERN ,q, MIR))
(TIN\FONT a IhllOERN a)
to,'F..H" ~)
hll!nEr.H 6

iBIrFnNT j (;',nPF~N 1P BFR)
'!HE".'LETIcA lG BRF)
hlrPEF;11 16 ~RP)

iTE.\TFrNT r '.,6LM"~'.lC 13)
'iTIhlE:'Pnn,,"N In)
i.LL~.~:IC lot)
!'TE\TBnLPFnNT
tCL~~CIC 16 Bfifi.,'
~TIME.';RL1MAN

1P BfiR)

tP:LAc.~,Ir 16 BRR]
[cT~NPARP (FDNTPrnPiLE
(PEF"ULTFnNT 1

Figure 31.8. Part of the value of the atom FONTDEFS
To install a new profile from this list, follow the following example, but insert any keyword for BIG.

To use the profile with the keyword BIG instead of the standard one, evaluate the following expressioh
(FOMTSET ‘BIG))

Now the fonts are permanently replaced. (That is, until another profile is installed.)

FONIS 319
1

r.

FUNCTIoNS FOR USING F0NTS

[[SMALL cFONTPROFlLE
(OEFALlLTPONT i (TERMINAL
6)

*U’acHA 6)
tøTERmIHAL 6))

(SOLPFL~NT (M’1.OERN 6 BRR)
tHELY’FTIL"’ 6 BRR)
Ihll!OER’H qL BRR))
i LITTLEFCNT ~"
(MlcERN 6 MIR)
lHEL’v’ETIu’"’ 6 MIR)
iMCiOERN ,qø MIR))
(TIN\FONT a IhllOERN a)
U,,F.,H" aj
hll!nEr.N 6

iBIrFnNT J ‘;;1nPF~N 1P BFR)
‘!HE".’LETICA 16 BRF)
hlrPEF;i1 16 ~fiP) !‘

i TE.\TFrNT r 6L"~.’lc 1’~)
liTIhlE;:pnMN In)
i.LL~.>:Ic In:)
!‘TE\TBnLPFnNT
t CLA~C 1 16 Bfifi
jTIME.’;ROMAN
1P BfiR)

\P:LAc.~.Ir 16 BRR]
[<~T~NPARP (FlNTPRnPILE
(PEF"ULTFnNT 1

Figure 31.8. Part of the value of the atom FONTDEFS
To install a new profile from this list, follow the following example, but insert any keyword for BIG.

To use the profile with the keyword BIG instead of the standard one, evaluate the following expressioh

(FlTSET ‘BIG))

Now the fonts are permanently replaced. (That is, until another profile is installed.)

FONTS 319
1

----- Next Message -----

Date: 19 Dec 91 16:35 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.163540pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11681>; Thu, 19 Dec 1991 16:35:49 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:35:40 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

12. YOUR INlT FILE

Interlisp-D has a number of global variables that control the environment of your 1108 or 1186. Global variables make it easy to customize the environment to fit your needs. One way to do this is to develop an "INlTø’ file. This is a file that is loaded when you log on to your machine. You can use it to set variables, load files, define functions, and any other things that you want to do to make the Interlisp-D environment suit you.

Your lnit fi’e could be callecl INlT, INlT.LlSP, INlT.USER, or whatever the convention is at your site. There is no default name preferred by the system, it just looks for the files listed in the variable USERGREETFILES, (see below). Check to see what the preference is at your site. Put this file in your directory. Your directory name should be the same as your login name.
The INlT file is loaded by the function GREET. GREET is normally run when Interlisp-D is started. If this is not the case at your site, or you want to use the machine and Interlisp-D has already been started, you can run the function GREET yourself. If your user name was, for example, TURlNG, then you would type:
(GREET ‘TURIK)

This does a number of things, including undoing any previous greeting operation, loading the site init file, and loading your init file. Where GREET looks for your INlT file depends on the value of the variable USERGREETFiLES. The value of this variable is set when the system’s SYSOUT file is made, so check its value at your site! For example, its value could be:
- . - 11
NIL

3’USERGREETFlLE5

iiiFD5hl,(.LI5PFILES~ USER ;INIT.LISPJ
t1rD5h’,’.LI5PFILE.>.~INIT.LI5PJ
t’,rFLoPPY’,INIT.L15žJ

i,rosh’,’øLI5PFILES\ USER .‘INIT.U5ERJ
((O.h L FILE.’ .INlT.U.’ER’øj

i(D. . FIL SER INIT:,
i(FLUPP';'j I

F~ure12.1. ApcsstblevalueofUSERGREETFILES.
In each place you see, "> USER >", the argument passed to GREET is substituted i:ito the path. This is your login name if you are just starting Interlisp-D. For example, the first value in the list would have the system check to see whether there was a file, [DSX]<L'SPFlLES>TURlNG>lNlT.LlSP. No error is generated if you do not hcve an INlT file, and none of the files in USERGREETFZLE$ are found.

Y0UR NIT FILE 12 1

MAKING AN INlT FILE

12.1 Making an lnit File

As described in Section 11.5, Page 11.7, each lnterlis~D program file has a global variable associated with it, whose name is formed by appending "COMS" to the end of the root filename. For any of the standard INlT file names, the variable INlTCOMS is used. To set up an init file, begin by editing this variable. First, type:

(SETQ I*ITco*s '((VAnS)))
Now, to edit the variable, type:
(l z:sicn*s>

A DEdit window wiil appear. This DEdit window is the same as the one called with the function OF, and described in Section 11.3, Page 11.4. This chapter will assume that you know how to use the structure editor, DEdit.

The CONS variable is a list of lists. The first atom in each internal list specifies for the file package what types of items are in the list, and what it is to do with them. This section will deal with three types of lists: VARS, FILES, and P. Please read about others in the lnterlis~D Reference Manual, Volume ll, Chapter 17.
The list that begins with "VAR5'ø allows you to set the values of variables. For example, one global variable is called DEditLinger. Its default value is T, and means that the Dždit window won't close affer you exit DEdit. If it is set to NIL, then the DEdit window will be closed when you exit DEdit. To set it to NIL in your INlT file, edit the VARS.list so that it looks like this:
ø . . .1 1 ø1
((',4R.$' iOEdirLinger NlLii Her
B~,are
G~lete
Replace
'3yvitch
( )

(out
Undo
Find
5'rtap
Rcprint
Edit

EditCam
Break
Eva

Exit

Figurø 12J. Setting the variable DEdI tLi nge r in INITCONS.
Notice that inside the vars list, there is yet another list. The firtt item in the list is the name of the variable. It is bound to the value of the second item. There are many other variables that you can set by adding them to the VARS list. Some of these variables are described in Chapter 43, and many others can be found in the lnterlis~D Reference Manual.

If you want to automatically load files, that can be done in your init file also. For'exampe, if you always want to load tho Library file SPY. DCOM, you can load it by editing tho INlTC0MS variable to list the appropriate file in thø list starting with FILES:

12.1 YOUR NIT flu

MAKING AN INlT FILE

(yARS iflEdlr.Llngør NIL') After
ff1LE~ _ ~PY\) Betott
Delete
Replace
Switch

()out
Undo
Find
Swap
Reprint
Edit

EddCom
Breok
Evol
Exit

FluFe 12.3. INITCOMS changed to load the file SPY.DCOM
Other files can also be added by simply adding their names to this FILES list.

Another list that can appear in a COMS list begins with "P". This list contains Interlisp-D expressions that are evaluated when the file is loaded. Do not put DEFINEQ expressions in this list. Define the function in the environment, and then save it on the file in the usual way (see Section 11.6, Page 11.7).
One type of expression you might want to see here, however, is a F0NTCREATE function (see Section 31.2, Page 31.2). For example, of you want to use a Helvetica 12 BOLD font, and there is not a fontdescriptor for it normally in your environment, the appropriate call to FOffTCREATE should be in the 'øP" list. The INlTCOMS would look like this:

.
((VARS (DžditLingcr NIL)) After
(FILES SPY) Betone
(~ JFoHTcREaTE (QUOTE Delcte
HEL\'ETIl',, Repace
~vyitch

1-
~juoTE _ SOL") _ .1)) ('out
Undo
Find

Swap
Reprint
Edit

EdiKom
Break
Eva
Exit

Figure 12.4. ltulTcOfl5editedtoincludeacalltofOffTCflEATE. The form will be evaluated when thelNlT file is loaded.

To quit, exit from DEdit in the usual way. When you run the function NAKEFiLES (See Section 11.6, Page 11.7.), be sure that you are connected to the directory (see Section 8.7, Page 8.4) where the INlT file should appear. Now when GREET is run, your init file will be loaded.

Y0UR INlT FILE 123

----- Next Message -----

Date: 19 Dec 91 16:48 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.164812pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11686>; Thu, 19 Dec 1991 16:48:22 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:48:12 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

r 33. MASTERSCOPE

Masterscope is a tool that allows you to quickly examine the structure of complex programs. As your programs enlarge, you may forget what variables are global, what functions call other functions, and so forth. Masterscope keeps track of this for you.

Suppose that JVTO is the name of a file that contains many of the functions involved in a complex system and that LlNTRANS is the file containing the remaining functions. The first step is to ask Masterscope to analyze these files. These files must be loaded. All Masterscope queries and commands begin with a period followed by a space, as in
ø AliLYZE FKS a Jvro

The ANALYZE process takes a while, so the 5ystem prints a period on the screen for each function it has analyzed. (See Figure 33.1)

82&. ANALYZE FNS ON 3VTO
. d.,ne
D3~. aNALY?E FNS ON LIH1R’N~

. 1a,lA.l

Figure 33.1. The Interlisp-D Executive Window affer anolyzing the files
If you are not quite sure what functions were just analyzed, type the file’s CONS variable (See Section 11.5, Page 11.7.) into the Interlisp-D Executive Window. The names of the functions

stored on the file will be a part of the value of this variable.

A variety of commands are now possible, all referring to individual functions within the analyzed files. Substantial variation in exact wording is permitted. Some commands are:
ø SHoN PATHS FRDN ANY T0 ANY
ø EDIT WERE ANY CALLS functionname
ø EDIT WERE ANY USES variablename
ø Wo CALLS WDN

ø Wo CALLS functionname
ø BY WoN IS functionname CALLED
ø WD USES variablename AS FIELD

Note that the function is being called to invoke each command. Refer to the /nterlisp-D Reference Manual for commands not listed here.

Figure 33.2 shows the lnterlis~D Executive Window affer the commands ø wno CALLS GobbleDunp and ø vffo DOES JVLinScan CALL.

MASTb'R'j~OPE 331

MASTEH,COPE

NIL

7,.,'. 1,,.lillj O~LL;==: ,1)~8 iD.B~imp

(,"c.h.~t,:r~i'TJ .J;/,j~J,J .J'.'t'r'Jet'TJ J;,'~~ 1Tij Gi>"ri'.'p~" ,,)bbl,,Ffu:h ,"jbb1~'Srririll
I/dump Fiji

'...,9j', "Ho clclE.. .J"i'L i '-. r, 1'"'LL
(Liri.'ci-ri 1'ø.'Cfr.3b1A 3 -h1~J
'9'A

Figure 33.2. Sample Masterscope Output

33.t The SHOW DATA command and GRAPNER

When the library package GRAPHER is loaded, (to load this package, type (FILESLOAD GRAPHER).) Masterscope's SHOWPATHS command is modified. The command will be changed to generate a tree structure 5howi ng how the program's functions interact instead of a tabular printout into the lnterlis~D Executive window. For example, typing:
ø ~ PATHS FW Proce:sžE.
produced the display shown in Figure 33.3.

.GtB.,31nT~, T L:.n;.lLl:'ø.Utøn:P,.=
~"''-,Jt",,;r,.pj

r~infr.:p it'~ø;r.!rPr:p
r.>~>(Li;'7Uiøn~p..:
..~..JtL,;.l. Ofl'.JtLl;l.
. ',r:L,st lET _ p-:
ž::J8:~inEnJ ,*.l.Tø.'r:.

ø .Err.'.r Pfl!I Pr'ni.~n,.;
:p~1y
~lnt~nlno

Figurø 33.3. SHOW PATHS Dsplay Example

All the functions in the display are part of this analyzed file or a previously analyzed file. Boxed functions indicate that the function name has been duplicated in another place on the display.

Selecting any function name on the display will pretty print the function in a window. (See Figure 33.4.)

ij.J MAsTERscopa

THE SHOW DATA COMMAND AND GRAPHER

-~&lLir1wilhS høfi
.ø~Tløo1nTwTr1no~
~9i"~

ø 1n,fl~ _ ~i~or9~~~
ø ~~&tl1r?inith. "(s
.

to~~tLisi _ ~~otLirt
~ø~.~.~rø,ø; ______
Pw:øLlrt ~(LøTTø.’

ø .‘ .d.~~~~;ø ~qLT~’ f

ø øø..‘PøintError PTl.I Frint~ninøi
ø

ø .- upv
~inlWr,r.

[LAnaPA ~propnaaoø’i (‘ cdttod: øø16øMAAøø3ø’ L’6’’døø
ø øłeCAn~’9SCorProp prcpneae (suR 1012C8L’øk])

Figutt 33.4. Browser Printout Example.

Selecting it again with the leff mOuse button will produce a dexription of the function’s role in the overall system (See Figure 33.4)

~r.ø;l.1,’?U1ønø;øf’.:ø
øt1BfgøinTW:øl.riny~ ________

,p’~ø~ 1"~ø:t,-’øøø.Pr’~
. ~c.3v.Liøiniih~t

Proc&;:Eli<. _________
ø .. . Por;ø~ør’ ø~:l=T7øoø.

ønf&rø.øør ~T=i Prir,t.~nir,ø
ø øPøntWr,1fl4ø

ø GerryProp i,, -
ø L-Qll:! inetAnC .rorPrtihø
1nžNžl,~ lrireFiøJ.,rningø
Din" ‘pe~øbøø’c8cJ1nT;.,=’øtr1n0.
i=~rMLø,ø 1rø’"ø$øT=~’,Før=Ceø’øøEtlD
ø u~’ f.-ccø TO cocl

FløUrf 33.5. Browser Description Example.

33.2 Databasefns: Automatic Construction and Upkeep of a Masterscope Database

DataBaseFns is a separate library package that allows you to automatically construct and maintain Masterscope databases of your filesø The package is contained in the DATABASEFNS.DCOM file.

When DATABASEFNS.DCOM is loaded, a Masterscope database will be automatically maintained for every file whose.DATABASE

DATABASEFNS: _ AUTOMATIC CONSTRUCTION AND _ UPKEEP OF A MASTERSCOPE _ DATABASE

property has the value YES. If this property's value is not set, you will be asked when you save the file "Do you want a Masterscope Database for this file?". Saying YES enables the DabaBaseFns to construct a Masterscope database of the file you are saving.
Each time the function *AKEFILE is used on a file whose DATABASE property has a value YES, Masterscope will analyze your file and update its own database. Each file's masterscop database is kept in a separate file whose name has the form FILE. DATABASE. Whenever you load a file with a YES value for its DATABASE property, you will be asked whether you also want the database file loaded.

----- Next Message -----

Date: 19 Dec 91 16:50 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.165058pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11688>; Thu, 19 Dec 1991 16:51:02 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:50:58 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

rø 34.WHERE DOES ALL THE TIME GO?

sPY

SPY is an Interlisp-D library package that shows you where you spend your time when you run your system. It is easy to learn, and very useful when trying to make programs run faster.

34.1 Now to use Spy with the SPY Window
The function SPY. BUTTON brings up a small window which you will be prompted to position. Using the mouse buttons in this window controls the action of the SPY program. When you are not using SPY, the window appears as in Figure 34.1.

Figure 34.1. The SPY window when SPY 15 not being rnsed.
Ts use SPY, click either the leh or middle mouse button with the mouse cursor in the SPY window. The window will appear as in Figure 34.2, and means that SPY is accumulating data about your program.

Figure 34.2. The SPY wir.oow when SPY is being used

To turn off SPY affer the program has run, again click a mouse button in the SPY window. The eye closes, and you are asked to position another window. This window contains SPY's results. An ex~nr'ple of result window is shown in Figure 34.3.

HOW TO USE SPY WITH THE SPY WINDOW

- TIrE.

l _ '3~~"H[P _ J~. _ [WIT. _ IN~&F.-1!~

17 _ '..TIrtP. PplJl'.E,-'-'..
- a ~i~~~~pT~ž&-

. REPE,,TE&L'.EV~rn EJ~rn ø1 EF.žUFE

7 _ ---RR.. h.JPillU~. _ fPlLE-ø -. 4 f, IPP,9R.h 'F..n 4
Figure 34.3. The window produced afler running $PY

This window i5 scrollable in two directionsø horizontally, and vertically. This is useful, since the whole tree does not fit in the Winoovv. If a part that you want to see is not shown, then you can scroll the window to show the part you want to see.

34.2 How to use SPY from the Lisp Top Level
SPY can also be run while a specific function or system is being used. To do this, type the function WITH. SPY:
(VITN.sPY form)

The expression used for form should be the call to begin running the function or system that SPY is to watch. If you watch the SPY window, the eye will blink! To see your results, run the function SPY. TREE. To do this, type:

(SPY.TREE)

The results of the last running of SPY will be displayed. If you do this, and 5PY.TREE returns (no SPY saiples have been gathQ red), your function ran too fast for SPY to follow.

34.3 Interpreting SPY's Results
Each node in the tree is a box that contains, first, the percentage of time spent running that particular function, and second, the function name. There are two modes that can be used to display this tree.

The default mode is cumulative. In this mode, each percentage is the amount of time that function spent on top of the stack, plus the amount of time spent by the functions it calls.
The second mode is individual. To chango the mode to individual, point to the title bar of the window, and press the middle 'n.ouse button. Choose Individual from the menu that appears. In this mode, the percentage shown is the amount of time th3t the function spent on the top of the stack.

lNTERPREn~ SPY'S RESuLtt

----- Next Message -----

Date: 19 Dec 91 16:40 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.164041pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11685>; Thu, 19 Dec 1991 16:40:51 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:40:41 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

32. THE INSPECTOR

The Inspector is a window-oriented tool designed to examine data structures. Because Interlisp-D is such a powerful programming environment, many types of data structures would be difficult to see in any other way.

32.1 Calling the Inspector

Take as an example an object defined through a sequence of pointers (i.e. a bitmap on the property list of a window on the property list of an atom inaprogram.)
To inspect an object named NAME, type:
(IKSPECT ‘~)

If NAME has many possible interpretations, an option menu will appear. For example, in Interlisp-D, a litatom can refer to both an atom and a function. For example, if NAME was a record, had a function definition, and had properties on its property list, then the menu would appear as in Figure 32.1.

PRG’PS
FklS

FIELD;=~

Figure 32.1. Option Window For Inspection of NAME

If NAME were a list, then the option menu shown in Figure 32.2 would appear. The options include:
ø calling the display editor on the list;

ø calling the ~ editor (the "Typing Shortcuts",Chapter 6);
ø seeing the list’s elements in a display window. If you choose this option, each element in the list will appear in the right column of the Inspector window. The left column of the Inspector window will be made up of numbers. (See Figure 32.3.)

ø inspecting the list as a record type (this last option would produce a menu of known record types). If you choose a record type, the items in the list will appear in the right column of the Inspector window. The left column of the Inspector window wili be made up of the field names of the record.

P~rI~.lErtr
Tr:rE’1ir.
Inip~rr

A~are’iJrd

Figure 32.2. Option Window For Inspection of Lirt

32.2 Using the Inspector

If you choose to display your data structure in an edit window, simply edit the structure and exit in the normal manner when done. If you choose to display the data structure in an inspect window, then follow these instructions:

ø To select an item, point the mouse cursor at it and press the left mouse button.

ø Items in the right column of an Inspector window can themselves be inspected. To do this, choose the item, and press the center mouse button.

ø Items in the right column of an Inspector window can be changed. To do this, choose the corresponding item in the left column, and press the center mouse button. You will be prompted for the new value, and the item will be changed. The sequence of steps is shown in Figure 32.3.
ø .

1 INPEu’T-ME-TOOl
1ie-,PErT-hlž-TQi32

a IN.u’FErT-11E-TQO3 The item in the lefl column is selected, and the middle mouse button pressed. Select the SET option from the menu that pops up.

The ev..pre:1Un re3J will be E
‘/~LuQred.

cHaflGE&-’.:~"LlJ4

1]N.=~Pfi=.T-rrtE-Tc~i
2 1H".~pEcT-ttE-TI:i12

a Il You will then be prompted for the new value. Type it in.

ø6

1 [flPEQT-ME-TOOi
2 [Y.~PECT-1’1E-TOfl2

a CH~Pl,’ED-’.;~LUE The item in the right column is updated to the value of what you typed in.

Figure 32.3. The sequence of steps involved in changing a value in the right column of an Inspector window.

32.3 Inspector Example

This example will use ideas discussed in 5ection 37.1. An example, ANlMALGItAPH, is created in that section. You do not need to know the details of how it was created, but the structure will be examined in this chapter.

If you type

(IKSPECT lI~.6liPN)

and then choosø thø Inspect option from thø menu, a display appøars as shown in Figure 32.4. ANlMAL.G~PH is being

J

33.J TkElNSPECT~

lff5PE~0R EXAMPLE

inspected as a list. Note the numbers in the left column of the inspectorwindow.

1 i’t’fI.~H ~ NIL NIL --j ‘BIRD .~ NIL NIL
ø T
.i, NIL
4 NIL
5 NIL
6 NIL
? NIL
,qø NIL
9. NIL
1A. NIL
11 NIL
1~" NIL

Figure 32.4. Inspector Window For ANIMAL GRAPH, inspected as a list.
If you choose the "As A Record" option, and choose "GRAPH" from the menu that appears, the inspector window looks like Figure 32.5. Note the fieldnames in the leff column of the inspectorwindow.

UP"PH.CH"NCEL"eELFfl NIL
CR"PH. INVEP.TL~BELFN NIL
CR"PH. IFlvEp.TBCiROERFN NIL
CR"PH.FONTcH"NoEFN NIL
bRaPH.&ELETELINKFN NIL
CRaPHø~D&LINkFN NIL
URAPH.cžLETENC~UEFN HIL
bRAPH. .oo&NUGEFN NIL
oRoPH.Mo$ENUoEFN NIL
DIREcTEDfLG NIL
o"IDE~FLo T

C.RuPHNi:DE.~ (i.’fl:H & NIL NIL --! ‘BIPP & NIL GIL

Figure 32.5. Inspector Window For ANlMAL.GRAPH, inspected as an instance of a "GRAPH" record.

The remaining examples will use ANlMAL.GRAPH inspected as a list. When the first item in the Inspector window is chosen with the leff mouse button, the Inspector window looks like Figure 32.6.

1 ‘ _ ø1
T
3 NIL
4 NIL
5 NIL
r~ NIL
NIL

NIL
9 NIL
1H NIL
11 NIL
1- NIL

Figure 32.6. Inspector Window For ANlMAL.GRAPH With First Element Selected
When you use the middle mouse button to inspect the seiected
list element, the display looks like Figure 32.7.

INSPECTOR EXAMPLE

1 ø1
T

3 NIL =
4 PIlL

5 NIL 1 iFIfl 1.19:’ 44) PIlL NIL HIL --!
‘BIRD (.192 29) NIL PIlL NIL --
b’ NIL 3 (CAT (.is ,J NIL NIL NIL

PIlL j. (&UU i"1;39 7) PIlL PJIL NIL
NIL ~ ((rh,,"trtffi,,"L GJU c~T) 199 14j fiL J.IlL
9 PIlL 6 ((,,"PIIMAL ; BIRD FI.Jh) .‘..~ C9. IlL
19 PIlL
11 NIL
1’.’ NIL

Figuro 32.7. Inspertor Window For ANlMAL.GRAPH and For the First Element of
ANIMALøGRAPH

How you can see that 5iX items make up the list, and you can
further choose to inspect one of these items. Notice that this is
also inspected as a list. As usual, it could also have been
inspected as a record.

Select item 5 - MAMMAL DOG CAT - with the leff mouse button.
Press the middle mouse button. Choose "Inspect" to inspect
your choice as a list. The Inspector now displays the values of the
structure that makes up MAMMAL DOG CAT. (See Figure 32.8.)

1 (h1~~MMkL GJ, lIT)
2 ilvjy’ lJ)
NIL
4 NIL
5 NIL
6 45
7

is

o i’,Do’: ClIT’,i

!) (c"’NlMlIL .~ BlRP FI."3Hj
iR, (Fi=1NTCLn"’,~r’j7R!i?e..764
ii hllIPtMlIL
12 NIL

Figure 32.8. Inspector Window for Element S From Figure 32.7 That Begins
((MAMMAL DOG CAT).

32.A THE INSPECTOR

----- Next Message -----

Date: 19 Dec 91 16:54 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.165444pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11690>; Thu, 19 Dec 1991 16:54:53 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:54:44 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

. . . .

rø 34.WMERE DOES ALL THE TiME GO?

sPY

SPY is an InterlispøD library package that shows you where you spend your time when you run your system. It is easy to learn, and very useful when trying to make programs run faster.

34.1 How to use Spy with the SPY Window
The function SPY. BUTTON brings up a small window which you will be prompted to position. Using the mouse buttons in this window controls the action of the SPY program. When you are not using SPY, the window appears as in Figure 34.1.

Figure 34.1. The SPY window when SPY is not being used.
Ts use SPY, click either the Iefl or middle mouse button with the mouse cursor in the SPY window. The window will appear as in Figure 34.2, and means that SPY is accumulating data about your program.

sPY

Figure 34.2. The SPY wirdow when SPY is being used

To turn off SPY atter the program has run, again click a mouse button in the SPY window. The eye closes, and you are asked to position another window. This window contains SPY's results. An example of result window is shown in Figure 34.3.

WHERE D0ES ALL THE TIME Go' SPY 341
l

How TO USE SPY 'KlTH THE SPY WINDOW

rp.i)rE?
~L,:,,.*~.

IrtP.rpji=.E;'';. .

øU TI~REhpYfiž&.

J!l!i .EV~fi)f. ':. ø ø1 FEPEA~OL.EU~rn -'1 EJ~J .l ER.GURE

7 _ øøø.BN.ø.F i;f;iø.iU~. _ Fpi'cføø:ø11 .. j IPP,fl~.øhJri.øii.iN

4

Fiqure 34.3. The window produced affer running SPY

Tljis window i5 scrollable in two directions, hOrizontaily, and vertically. This is useful, since the whole tree does not fit in the wiroow. If a part that you want to see is not shown, then you can scroll the window to show the part you want to see.

34.2 How to use SPY from the Lisp Top Level
SPY can also be run while a specific function or system is being used. To do this, type the function VITH SPY:
(WITH.sPY form)

The expression used for form should be the call to begin running the function or system that SPY is to watch. If you watch the SPY window, the eye will blink! To see your results, run the function SPY. TREE. To do this, type:

(SPY.TREE)

The results of the last running of SPY will be displayed. If you do this, and SPY.TREE returns (no SPY saiples have been gathered), your function ran too fast for SPY to follow.

34.3 Interpreting SPY’s Results
Each node in the tree is a box that contains, first, the percentage of time spent running that particular function, and second, the function name. There are two modes that can be used to display this tree.

The default mode is cumulative. In this mode, each percentage is the amount of time that function spent on top of the stack, plus the amount of time spent by the functions it calls.
The second mode is individual. To chango the mode to individual, point to the titlo bar of the window, and press the middle .~ouse button. Choose Individual from the menu that appears. In this mode, the percentage shown is the amount of time that the function spent on the top of the stack.

1NTERPREfl~ SPY’S RESULTS
To look fit G iingle branch of the tree, point with the mouse curtor at one of the nodes of the tree, and press the right mouse hutton. From the menu that appeatt, choose the option SubTree. Another SPY window will appear, with just this branch of the tree in it.

Another way to focus within the tree is to remove branches from tlie tree. To do this, point to the node at the top of the branch you would like to delete. Press the middle mouse button, and choose Delete from the menu that appears.

There are also different amounts of "merging" of functions that can be done in the window. A function can be called by another function more than once. The amount of merging determines where the subfunction, and the functions that it calls, appear in the tree, and how offen. (For a detailed explanation of merging, see the Lisp Library Packages Manual.)

----- Next Message -----

Date: 19 Dec 91 16:59 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.165929pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11691>; Thu, 19 Dec 1991 16:59:33 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 16:59:29 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

tilL.. 36. FREE MENUS

Free Menu is a library package that is even more flexible than the regular menu package. It allows you to create menus with different types of items in them, and will format them as you would like. Free menus are pai~icularly useful when you want a "fill in the form" type interaction with the user.
Each menu item is described with a list of properties and values. The following example will give you an idea of the structure of the description list, and some of your options. The most commonly used properties, aiid each type of menu item will be described in Section 36.2 and Section 36.3.

36.1 An Example Free Menu

Free menus can be created and formatted automatically! It is done with the function FN. FORNATNENU This function takes one argument, a description of the menu. The description is a list of lists; each internal list describes of one row of the free menu. A free menu row can have more than one item in it, so there are really lists of lists of lists! It really isn’t hard, though, as you can see from the following example:

```
(SETQ Ex~1e*anu
(F*.FORliT*EMu

‘(( TYPE TITLE LABEL TitlesDonothing)
TYPE 3STATE LABEL Ex~1e3State))

( TYPE EDITSTART LABEL PressToStartEd;ting
ITEMS (EDITE*))

(TYPE EDIT ID EDITEN LABEL øø))
(*IKDDMPRDPS TITLE øEx~1e Dris Nothing))))
```

The first row has 2 items in it; one is a TITLE, and the second is a 35TATE item. The second row also has 2 items. The second, the EDIT item, is invisible, because its label is an empty string. The caret will appear for editing, however, if the EDlTSTART item is chosen. Windowprops can appear as part of the description of the menu, because a menu is, affer all, just a special window. You can specify not only the title with WINDOWPROPS, but also the position of the free menu, using the "Ieff" and "bottom" properties, and !he width of the border in pixels, with the "border" property. Evaluating this expression will return a window. You can see the menu by using the function OPENW. The following example illustrates this:

AN EXAMPLE FREE MENU

6i,'~T~ E;mD.1c1d~nij

.;F(,7,fJp,[4,,$~]~fJ.J\J ,, ø. T' Gf TITLE LBEL T ir1~,flN~rr T
.T"rE =-ø'T."TE L"bEL E:.Jm"1c5tJcs'!'.
. 'FE =,IT:..THF:T

LEL =r~'=Tu'"r~t'tEditing
lTEfl= cOlTEN'

T"E 'IT ID EDITEm L~8EL ,
"'..ililci=,..''=cipT=..

TITLE ' ;,.,1";c Din' 1'luthlnj.'?

.TT9'i i)pf)liff molMertiJ'i
f.hi1ltlDu',V'r#' j64

Figure 36.1. An example free menu

The next example shows you what the menu looks like affer the EDlTSTART item, PressToStartEditing, has been chosen.

T,r f~"=.Oi=i1'1,=irhin3 E':,mp1~,='.=r.,r~
P~'~='"'TJT..,rTEJ1r1r1.j A

Figure 36.2. Free menu affer the EDlTSTART item has been chosen
The following example shows the menu with the 3STATE item in its T state, with the item highlighted (In the previous bitmaps, it was in its neutral state.)
.
ø c l 1 1

.1-=':.OiJ-tljrhini=!
,:'T'='Ot..;rrE'liriiJ,

FigUre 36.3. Free menu with the 35TATE .tem in its T state
Finally, Figure 36.4 shows the 35TATE item in its NIL state, with a diagonal line through the item

T1r le.".OcNorhing E..::r'~ _ 1 _ = _ ø'.'..,i.~
Rrn. ;'', T,St.arrEdir,iri,

. . . . .

Figure 36.& Free menu with the 3STATE item in its NIL state
If you would like to specify the layout yourself, you can do that too. See the Lisp Library Packages Manual for more information.

36.2 Parts of a Free Menu Item
There are 8 different types of items that you can use in a free menu. No matter what type, the menu item is easily described by a list of properties, and values. Somo of the properties you will use most often are:

LABEL Required for every type of menu item. It is the atom, string, or bitmap that appears as a menu selection.

TYPE One of eight types of menu items. Each of these are described below.

MESSAGE The message that will appear in the prompt window if a mouse button is held down over the item.

ID An item's unique identifier. An ID is needed for certain types of menu items.

ITEMS Used to list a series of choices for an NCHOOSE item, and to list the ID's of the editable items for an EDITSTART item.
SELECTEDFN The name of the function to be called if the item is chosen

36.3 Types of Free Menu Items
Each type of menu item is described in the following list, including an example description list for each one.

Momentary This is the familiar sort of menu item. When it is selected, the function stored with it is called. A description for the function that creates and formats the menu looks like this:
(TYPE WEKTARY
LABEL Blink-K-Rin9

*ES~6E øBlinks the screen and rings bellsø
sžLEcTEDFK RIKBELLS)

TOGGLž This menu item has two states, T and NIL. The default state is NIL, but choosing the item toggles its state. The following is an example description list, without code for the SELECTEDFN function, for this type of item:
(TYPE T~6LE
LABEL hi~isab1e

sELEcTEDFN changeIl*State)

3STATE This type of menu item has 3 states, NUIETRAL, T, AND NIL Neutral is the default state. T is shown by highlighting the item, and NIL is shown with diagonal lines. The following is an example description list, without code for the SELECTEDFN function, for this type of item:
(TYPE 3STATE

LABEL correctprograøAllofflospelling
sELEcTEDFli ToggleSpellingcorrection)

TITLE This menu item appears on the menu as dummy text. It does nothing when chosen. An example of its description:
(TYPE TITLE LABEL øChoices:")

NWAY A group of items, nnly one of which can be chosen at a time. The items in the NWAøY group should all have an ID field, and the ID's should be the same. For exan1Fle, to set up a menu that would allow the user to chose betvveei Helvetica, Gacha, Modern, and Classic fonts, the descriptions might look like this (Once again, without the code for the SELECTEDFN):
(TYPE lAY ID F~Tc~Ic'
LABEL blvetica
sELEcTEDFN changeFont)

TYPES OF FREE MENU ITEMS

(TYPE NVAY ID FOQTCKICE
LABEL Gacha
SELECTEDF

(TYPE lAY ID F05TliCriC0ha,~n8efont)
LABEL Modern

SELECTEDFli Chan2eFont)
(TYPE KAY ID fONTCHOIC
LABEL Classic

SELECTEDFN Changefont)

NCHOOSE This type of menu item is like NWAY except that the choices are
given to the user in a submenu. The list to specify an NCHOOSE
menu item that is analogous to the NWAY item above might
look like this:
(TYPE MC~SF
LABEL FontChoices

ITEMS Helvotica Gacha Modern Classic)
SELECT DfK Changefont)

EDlTSTART When this type of menu itein is chosen, it activates another type
of item, an EDIT item. The EDIT item or items associated with an
EDlTSTART item have their lD's listed on the EDlT5TART's ITEMS
property. An example description list is:
(TYPE EDITSTART LABEL øFunction to add? ITEMS (Fn))
EDIT This type of menu item can actually be edited by you. It is often
associated with an EDlT5TART item (see above), but the caret
that prompts for input will also appear if the item itself is chosen.
An EDIT item follows the same editing conventions as editing in
Interlisp-D Executive window:
Add Characters by typing them at the caret.
Move the caret by pointing the mouse at the new position, and
clicking the leff button.

Delete Characters from the caret to the mouse by pressing the
right button of the mouse. Delete a character behind the caret
by pressing the back space key.

Stop editing by typing a carriage return, a Control-X, or by
choosing another item from the menu.
An example description list for this type of item is:
(TYPE EDIT ID Fn LABEL øø)

36.4 FREENEMus
1

----- Next Message -----

Date: 19 Dec 91 17:05 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.170545pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11694>; Thu, 19 Dec 1991 17:05:54 PST

Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 17:05:45 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

37. THEGRAPHER

37.1 Say it with Graphs

Grapher is a collection of functions for creating and displaying graphs, networks of nodes and links. Grapher also allows you to associate program behavior with mouse selection of graph nodes. To load this package, type
(FILESL~ GliPHER)

Figure 37.1 shows a simple graph.
i ‘iLk w.F."PH ‘N M’L.l;R"PH ‘NlM’øL r;P"Pffø’
,(.h,,lINGUY!:łw’,1513.y1
14’

-FIH

. NIM"L, BIRO

Figure 37.1. A Simple Graph

In Figure 37.1 there are six nodes (ANIMAL, MAMMAL, DOG, LAT, FISH, and BIRD) connected by five links.
A GRAPH is a record containing several fields. Perhaps the most important field is GRAPHNODžS - which is itszlf a list of GRAPHNODE records. Figure 37.2 illustrates these data structures The window on top contains the fields from the simple graph. The window on the bottoms an inspection of the node, DOG.

i9’1, I ET ø‘NI1’1,,L.CR~PH’.i
Ilvl,l = ‘-#"=9,1j~’j’,’3

GPPH.cr"ilLEL,,ø’BELFN ‘IL
ø 1’R"pH. Ili’ø!ERTLBELFN 1.lIL
ø H. Ifløø.øøERTBDPDEPFN tilL
H.FGtTi’.HNoEFPl 1ølIL

, rPø’PH,t.lL/EløllDEFbl IL
. OIRECTECFLi, (ilL

ø rp..’Pflbll) ~ I.F = , tilL III ‘.øB.IP.D NIL ff IL

. NOOEBOPOER ‘ilL
tiODEL,,’BEL loo
, , ‘tODEFONT ~‘FOIiT ‘
. .OtlffO&EO- It’lL -

. t,iODE’.~,lOTH ‘.4,
, IiUOEL6EL.’-H~OE øølIL
, NODELfiELBlTIrt,iøP øIIL
, I,iUDEPUITlClI.l in
NODE ID 300

Figure 37.2. Inspefling a Graph and a Node

The GRAPHNODE data structure i~ described by its text (NODElD), what goes into it (FROMNODES), what leaves it (TONODES), and other fields that specify its looks. The basic model of graph building is to create a bunch of nodes, then layout the nodes into a graph, and finally display the resultant graph. This can be done in a number of ways. One is to use the function NODECREATE to create the nodes, LAYOUTGRAPH to lay out the nodes, and SHOWGRAPff to display the graph. The primer shows you two simpler ways, but please see the Library Packages Manual for more information about these other functions. The primer's first method is to use SHOWGRAPH to display a graph with no nodes or links, then interactively add them. The second is to use the function LAYOUT5EXPR, which does the appropriate NODECREATES and a LAYOUTGRAPH, with a list.
The function SHQWGRAPH displays graphs and allows you to edit them. The syntax of SHOWGffAPH is

(~liPH graph window lefibuttonfn middlebuttonfn topjustiffflg alloweditflg copybuttoneventfn)
Obviously the graph structure is very complex. Here's the easiest way to create a graph.
~.6liPN III)

lS5~liPN P.6liPH ø5Y Sraphø KIL NIL NIL T)

Figurø 37.3. My Graph

37.2 THEGRAPHER .J

SAY IT WITH GRAPHS

You will be prompted to create a small window as in Figure Figure 37.3. This graph has the title My Graph.
Hold down the right mouse button in the wiridow. A menu of graph editing operations will appear as in Figure 37.4.

D;Ier~ Link
&=h~~n9e ib P.I
ljbel g,nill~.r
l&'bel l~.roø~.r
Dir~..ct~.i1
SIdPg
~ BoiødP.r

'h;~d"
'Tr"P

Figure 37.4. A Menu of Graph Editing Operations
Here's how to use this menu to:

Add a Node Start by selecting Add Node. Grapher will prompt you for the name of the node (See Figure 37.5.) and then its position.

Figure 37.5. Grapher prompts for the name of the node to add affer Add Node is chosen from the graph editing menu.

Position the node by moving the mouse cursor to the desired location and clicking a mouse button. Figure 37.6 ,hows the graph with two nodes added using this menu.

~irør-ri~tle
s~~'ondnod~

Figure 37.6. Two nodes added to MY GRAPH using the graph ed it.q.g menu

AddaLink Select Add Link from the graph editing menu The Prompt
window will prompt you to select the two nodes to be linked.
(See Figure 37.7.) Do this, and the link will be added.

o .

first-node
,.ccond-node

Figure 37.7. The Prompt window will prompt you to select the two nodes to link.

DeleteALink Select Delete Link from the graph editing menu. ThePrompt
window will prompt you to select the two nodes that should no
longer be linked. (See Figure 37.8.) Do this, and the iink will be
deleted.

r _ ‘rr-n>’;1~
;‘~"or,’j-nod;

FigUre 37.8. The Prompt window will prompt you to Seje~~ ‘.1 ryo nodes that
shouid no longer be linked.

Delete A Node Select Delete Node from the graph editing menu. The Prompt
window will prompt you to select the node to be aeleted. (See
Figure 37.9.) Do this, and the node will be deletea.

firs. r-nod"
,L’0fl’S1-fl0d~

Figure 37.9. The prompt to delete a node

Moving a Node Select "Delete Node" from the graph editng menu. Choose a
node pointing to the it with the mouse cursor, and pressing and
holding the leff mouse button. When you move the mouse
cursor, the node will be dragged along. When the node is at the
new position, release the mouse button to deposit the node.

The commands in this menu are easy to learn. Experiment with
them!

37.2 Making a Graph from a List

Typically, a graph is used to display one of your program’s data
structures. Here is how that is done.

LATOUTSEXPR takes a list and returns a GRAPH record. The
syntax of the function is

(UYWTSEXPR sexpr format ~xing font motberd
penonald fam;lyd)
For example:

(u~T10Q AKIliL.TREE ‘(MIlL (l’~ ~ CAT) BIli FISH))
AaIliL.6liN

b~YouTSE*PR AKIliL .TREE øHoRIZ0NTALi~)
(Eli N AHIliL.GliPN øNj Grøpbø NIL KIL a T)

This is how Figure 37.1 was produced.

37.3 Incorporating Grapher into Your Program
The Grapher is designed to be built into other programs. It can call functions when, for example, a mouse button is clicked on a node. The function SHOWGRAPff does this:

(~liPH graph window leflbuttonfn middlebuttonfn
topjusti~Rg alloweditflg copybuttoneventfn)

For example, the third argument to SHOWGRAPH, leftbuttonfn, is a function that is called when the lefl mouse buttoii 15 pressed in the graph window. Try this:
(DEFIKžQ (~.LEfT.BUTT0N.FUNCTI0N
(TNE.6liPHNooE THE.GliPH.wIN~)
(INSPECT TNE.6liPNNooE)))

(~liPH FlILY.61PN øInspoct&bla fiilyø
(F~TIK N".LEFT.BUTTa.FuNCTIo*)
liIL NIL T)

In the example above, liT.LEFT.BUTTON. FUNCTION simply calls the inspector. Note that the function should be written assuming it will be passed a graphnode and the window that holds the graph. Try adding a function of your own.

37.4 More of Grapher

Some other Library packages make use of the Grapher. (Note: Grapher needs to be loaded with the packages to use these functions.)

ø NASTERSCOPE: The Browser package modifies the Masterscope command, . SHOW PATHS, so that its output is displayed as a graph (using Grapher) instead of simply printed.
ø GRAPHZOOM: allows a graph to be redisplayed larger or smaller automatically.

THEGRAPHER 375

----- Next Message -----

Date: 19 Dec 91 17:11 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.171147pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11697>; Thu, 19 Dec 1991 17:11:55 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 17:11:47 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

41. RESOURCE MANAGEMENT

41.1 Naming Variables and Records
You will find times when one environment simultaneously hosts a number of different programs. Running a demo of several programs, or reloading the entire Interlisp-D environment from floppies when it contains several different programs, are two

examples that could, if you aren't careful, provide a few problems. Here are a few tips on how to prevent problems:

ø If you change the value of a system variable, ffENUHELDVAIT for example, or connect to a directory other than (DsK)<LlsPFlLEs>, write a function to reset the variable or directory to its original value. Run this function when you are finished working. This is especially important if you change any of the system menus.

ø Don't redefine Interlisp-D functions or CLl5P words. Remember, if you reset an atom's value or function definition at the top level (in the Interlisp-D Executive Window), the message (Some.Crucial.Function. Or. Variable redefined), appears. If this is not what you wanted, type UNDO immediately!

If, however, you reset the value or function definition of an atom inside your program, a warning message will not be printed.

ø Make the atom names in your programs as unique as possible. To do this without filling your program with unreadable names that noone, including you, can remember, prefix your variable names with the initials of your program. Even then, check to see that they are not already being used with the function BOUNDP. For example, type:

(~P øB&ckgroundhnu)

This atom is bound to the menu that appears when you press the leff mouse button when the mouse cursor is not in any window. BOUKDP returns T. BOUNDP returns NIL if its argument does not currently have a value.

ø Make your function names as unique as possible. Once again, prefixing function names with the initials of your program can be helpful in making them unique, but even so, check to see that they are not already being used. GETD is the Interlisp-D function that returns the function definition of an atom, if it has one. If an atom has no function definition, GETO returns NIL. For example, type:

(GEffl 'CAR)

A non-NIL value i~ returned. The atom CAR already has a function definition.

ø Use complete record field names in record FETCHes and REPLACEs when your code is not compiled. A Complete record field name is a list Consisting of the record declaration name and the field name. Consider the following example:

REC0RD N~ FIRST LAST))

SETQ Nyfflrn create Nlž FIRST 'John LAST '~ith))
FETCH (~ FIRST) OF Mylrn)

ø Avoid reusing names that are field names of Interlisp-D System records. A few examples of system records follow. Do not reuse these names.

RECORD RE6IOl (LEFT RoTTOl WIDTH NEIGHT))
RECORD POSITIK xC00RD
RECORD Ili6E~ BITliYCP00RD)))

ø When you select a record name and field names for a new record, check to see whether those names have already been used.

Call the function RECLOOK, with your record name as an aørgument, in the lnterlis~D Executive Window. (See Figure 41.1.) If your record name is already a record, the record definition will be returned; otherwise the function will return NIL.

- . 11

4..Oø:ø(fi.ECL)OY~ FB;’~ITiON)
!‘P\ECCPO
PO~1TI)N

[øT\L~’.lp:E)F;,P "So’1P~D!
(8Nfl (LlSTP O~TUM\
(NUl18žP~P !‘,C4l’~ D~TUfi1:))

(i\"(’~’Tžh1\\j (NUMBERž (CDR OurOIl]
5ik(P~ECLOUff N~,~P~~)
NIL
5~’~E

Figuvø 41.1. RECLOOK returns tbe record definition If ts argument is already declared as a record, NIL otne~ise.

Call the function FIELDLOOK with your new field name in the InterlispøD Executive Window. (See Figure 41 2.) If your field name is already a field name in another record, the record definition will be returned; otherwise the function will return NIL.

412 RESOURCE MANAGEMENT ø1

NAMING VARIABLES AND RECORDS
, 1
~.4’+(fIELOLOOft Y96COORD)
((RECORD
pO’e.ITION

(:~‘COORO \COORD)
[T’Y’PE"ø (~NP (LIvt~TP O4TUbl!
(NUblBERP (CAR D,iTUhl\
I:.NUMBERP ("OR D~TUftt]
(S\ø’~~.TEb1)\)

55~(FIELPLOPh .ip~;\
NIL
58-

Figure 41.2. FIELDLOOK returns the record definition if ItS argument Is already the field ofarecord.NILothe~ise

41.2 Some Space and Time Considerations

In order for your program to run at maximum speed, you must efficiently use the space available on the system. The following section points out areas that you may not know are wasting valuable space, and tips on how to prevent this waste.

Often programs are written so that new data structures are created each time the program is run. This is wasteful. Write your programs so that they only create new variables and other data structures conditionally. If a structure has already been created, use it instead of creating a new one.

Some time and space can be saved by changing your RECORD and TYPERECORD declarations to DATATYPE. DATATYPE is used the same way as the functions RECORD and TYPERECORD. (See Chapter 24.) In addition, the same FETCH and REPLACE commands can be used with the data structure DATATYPE creates. The difference is that the data structure DATATYPE creates cannot be treated as a list the way RECORDs and TYPEREC0RDs can.

41.2.1 Global Variables

Once defined, global variables remain until Interlisp-D is reloaded. Avoid using global variables if at all possible!
One specific problem arises when programs use the function 6ENSYff. In program development, many atoms are created that may no longer be useful. Hints:
ø Use

(DELDEF atomname ‘PKP)
to delete property lists, and
(DELDEF atomname ‘vARS)

----- Next Message -----

Date: 19 Dec 91 17:15 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.171603pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11560>; Thu, 19 Dec 1991 17:16:06 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 17:16:03 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

to have the atom act like it is not defined.

These not only remove the definition from memory, but also change the appropriate f 11 eCOffS that the deleted object was associated with so that the file package will not attempt to save the object (function, variable, record definition, and so forth) the next time the file is made. Just doing something like
(SETQ (arg at~nm) ‘~IE)

looks like it will have the same effect as the second DELDEF above, but the SETQ doesn’t update the file package.
ø If you are generating atom names with GENSYN, try to keep a list of the atom names that are no longer needed. Reuse these atom names, before generating new ones. There is a (fairly large) maximum to the number of atoms you can have, but things slow down considerably when you create lots of atoms.
ø When possible, use a data structure such as a list or an array, instead of many individual atoms. 5uch a structure has only one pointer to it. Once this pointer is removed, the whole Structure will be garbage collected and space reclaimed.

41.2.2 Circular Lists

If your program is creating circular lists, a lot of space may be wasted. (Note that many cross linked data structures end up having circularities.) Hints when using circular lists:
ø Write a function to remove pointers that make lists circular when you are through with the circular list.

ø If you are working with circular lists of windows, bind your main window to a unique global variable. Write window creation conditionally so that if the binding of that variable is already a window, use it, and only create a new window if that variable 15 unbound or NIL.

Here is an example that illustrates the problem. When several auxilIary windows are built, pointers to these windows are usually kept on the main window’s property list. Each auxilIary window also typically keeps a pointer to the main window on its property list If the top level function creates windows rather than reusing existing ones, there will be many lists of useless windows cluttering the work space. Or, if such a main window is closed and will not be used again, you will have to break the links by deleting the relevant properties from both the main window and all of the auxiliary windows first. This is usually done by putting a special CLOSEFli on the main window and all of its auxiliary windows.

41.2.3 When You Run Out Of Space
Typically, if you generato a lot of structure! that won’t get garbage collected, you will eventually run out of space. The important part ii being aNe to track down those structures and

the code that generates them in order tO become more space efficient.

The Lisp Library Package GCHAX.DCOM can be used to track down pointers to data structures. The basic idea is that GCHAX will return the number of references to a particular data structure.

A special function exists that allows you to get a little extra space 50 that you can try to save your work when you get toward the edge (usually noted by a message indicating that you should save your work and sysin a fresh Lisp). The GAINSPACE function allows you to delete non-essential data structures. To use it, type:

(liIKSPACE)

into the Interlisp-D Executive Window. Answer "N" to all questions except the followi ng.
ø Delete edit history
ø Delete history list.

ø Delete values of old variables.
ø Delete your MASTERSCOPE datadase
ø Delete information for undoing your greeting.
Save your work and reload Lisp as soon as possible.

----- Next Message -----

Date: 19 Dec 91 17:23 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.172334pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11625>; Thu, 19 Dec 1991 17:23:37 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 17:23:34 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

42. SIMPLE INTERACTIONS WITH THE CURSOR, A BITMAP, AND A WINDOW

The purpose of this chapter is to show you how to build a moderately tricky interactive interface with the various lnterlis~D display facilities. In particular how to move a large bitmap (larger than 16 x 16 pixels) around inside a window To do this, you will change the CURSORINFN and CURSOROUTFN properties of the window. If you would also like to then set the bitmap in place in the window, you must reset the BUTTOKEVENTFN. This chapter explains how to create the mobile bitmap.

42.1 An Example Function Using GETMOUSESTATE

One function that you will use to "trace the cursor" (have a bitmap follow the cursor around in a window) is GETNOUSESTATE. This function finds the current state of the. mouse, and resets global system variables, such as LASTM0USEX andLASTMOUSEY.

As an example of how this function works, create a window by typing

```
(SETQ EzMPLE.wIN~ (CREATEI))
```

into the Interlisp-D Executive window, and sweeping out a window. Now, type in the function

```
(DEFIKEQ (PRIKTC00RDS (V)

P~TPRI*T ø(ø ~TuouSEx ., . US~EY ø)ø)
BLocK)

6E~SESTATE)))
```

This function calls GETNOUSESTATE and then prints the new values of LASTNOUSEX and LASTMOUSET in the promptwindow. To use it, type

```
(WIKraoPRoP EXlPLE .ilI*~ ‘CURSD~EDFK ‘PRIaTC00RDS)
```

The window property CURSORffOVEDFN, used in this example, will evaluate the function PRINTCOORDS each time the cursor is moved when it is inside the window. The position coordinates of the mouse cursor will appear in the prompt window. (See Figure 42.1.)

AN EXAMPLE FUNcTION USING GETMOUSE5TATE

Figure 42.1. The current position coordinates of the mouse cursor are shown in the prompt window

42.2 Advising GETMOUSESTATE
For the bitmap to follow the moving mouse cursor, the function GETKOUSESTATE is advised. When you advise a function, you can add new commands to the function without knowing how it is actually implemented. The syntax for advise is

(RISE fn when where what)

fn is the name of the function to be augmented.
when and where are optional arguments. when specifies whether the change should be made before, affer, or around the body of the function. The values expected are BEFORE, AFTER, or AROUND.

what specifies the additional code.

In the example, the additional code, what, moves the bitmap to the position of the mouse cursor. The function GETNOUSESTATE will be ADVISEd when the mouse moves into the window. This will cause the bitmap to follow the mouse cursor. ADVISE will be undone when the mouse leaves the window or when a mouse button is pushed. The AOVISEing will be done and undone by changing the CURSORINFK, CURSOROUTFN, and BUTTONEVENTFK forthewindow.

42.3 Changing the Cursor

Ofif laot part of tho examplo, to give thø impression that a bitmap is dragged around a window, thø original cursor should disappear. Try typing:
(CURSOR (CURSORCRžRrt (6I~PCREAtt 1 l) 1 11

CHANGING THE CURSOR

into the lnterlis~D Executive Window. This causes the original cuttor to disappear. It reappears when you type
(CURSOR T)

When the cursor is invisible, and the bitmap moves as the cursor moves, the illusion is given that the bitmap is dragged around the window.

42.4 Functions for "Tracing the cursor"
To actually have a bitmap trace (follow) the cursor, the environment must be set up so that when the cursor enters the tracing region the trace is turned on, and when the cursor leaves the tracing region the trace is turned off. The function

Establish/Trace/Data will do this. Type it in as it appears (note: including the comments will help you remember what the function does later).

(DEFIKEQ Establish/Trace/rata

[LANR0 (ønd tracebiteap cursor/rightoffse~ cursor/heighteffse~ GCGA6P)

ø This functlri is colløcd to ostablish tha døti to tracø
the døsirod bitaapø øøndø is the øind~ in øhich the tracing
is to take place, łtracebitøap" is the

øcursor/rightoffsetø and ø~~~50~'~~~g~~0t~~~~~5~i~~1 b~~i~~t~g~~5
øhich dete~ine the hotspot of the tracing biteap.
As "cursor/heightof'set and øcursor/rightoffsetø increase
the cursor hotspot :ves up and to the right.
If GCGAGP is non-NIL, GcGAC øill be disabled.)

(PRoG NIL

(if (0R NULL ønd)
(NULL tracebitaap))

then (PLAYTUNž (LIST (CONS 1000 4000)))

(if ~&~&pRET~RN))
then (GC6A6))

ø Create a blank cursor.)

(SSEETTQQ ~8BrnUNNKKCTliURCS0ECRtiR(soBIRllNAø(CPCURRsoEARTcEREl,eT~loø)jwxc~~2~øj~
ø ø Set the CURSOR IN and 0UT FNS for ønd to the
Jolloeing:)

(*INroNPRoP ønd UTE CURSORINF
(FU TIrn SETUP/Tlic

(WINDoNPooP ønd~~TE CURSoRoUTFENNJJ
(FU TIoN UNTlicž/CURSOR))

(O ø To all",' the bita,ap to be set den in the øindw bY
pressing a "'ouse button, include this line.
0ther',,'ise, it is not needed)

(WINnoNPRop ønd (UTE RUTToNEVENTFN)
(FUNCTIoN PLACEIBITNAPIINIwINrGN))

Set up Global Variables for the tracing 9eratien)
(SETQ øTRAcElITNApø tracebiteap)
TQ øRIGNTTliCE'oFFSETø(oR cursor/rightoffse~ 0)

5>sfE~TQ øHEIGHTTRACEIoFFSETø(0R cursor/hei htoffset))
TQ ø0LQBIT~PPosITIoNø(BIfflPCREATE llNArnIOTN tracebitøap)

(SETQ øTliCCwfNDoNø rndj)) BITNApHEI6hT tracebiteap)))

SIMPLE INTERAcT'oNs WITH THE cuRsoR. A BJTMAP. AND A WINDOW 423
1

FUNCTIONS FOR "TRACING THE CURSOR'

When the function Establish/Trace/Data is called, the
functions SETUP/TRACE and UNTRACE/CURSOR will be installed
as the values of the window's WlNDOWPROP$, and will be used
to turn the trace on and off. Those functions should be typed in,
then:

(DEFINEQ SETUP/TRACE
[ADA ønd)

(O ø This function is and’s cuRSORIKFK.
It siepiy resoti thø last trace position and the current
tracing region. It also raadvises fiETNouSESTATE to perforn
the trace function after each call.)

(if øTRAcEBITNAPø

then SETQ łLAST-lPACE-XPO5ø -zOo
SETQ øLAST-TRACE-YPOSø -zoooł)

SETQ øvNDREGIaø (WINOaNPROP and (ATE REGI0N)))
WIN~flROP and (ATE TliCIK)
T)

ø :ake the cursor disappear)

CURSOR łBLANKTRACECURSORø)
ANISE QUOTE GEThOUSESTATE)
QUOTE AFTER)
IL

(QUOTE (TPACE/CURSOR]))

(Dt~EQ(UaNTRACE/CURSOR

ø Th1s function is ønd’s CURsOROUTFN.
The function first checks if the cursor is currently being
traced; if so, it replaces thø tracing biiaap aith ahat is
under it and then turns tracing off by unadvislng
6ETNOUSESTATE and setting thø TliCIK ainda propertj of
łTRACEWINOoOø TO NIL.)

(if (VIN~PnOP øTRACEWINOONł(QUOTE TliCIK))
then (BITBLT ø0LOBITNAPPOSITIONø o o (scREENBIllNAP)
IPLUS CAR łffDRE6IONø)øLAST-lRACE-xPOSø)
(IPLUS 1CADR ø:DREGIOffo)øusT-TlicE-YPosø))
(WINOoePRoP ølliCEMINOONø(QUOTE TRACIK)
NIL))

replace the original cursor shape)
(CURSOR T)

unadvise 6E~sESTATE)
(U~"’ISE (QUOTE 6ETNOUSESTAlE]))

The function SETUP/TRACE has a helper function that you must
also type in. It is TRACE/CURSOR:
(DEFINEQ (TliCE/CURSoR
[LANRli NIL

ø This functi: does thø actual BITBLTln of thø tracing
blteap. This functla Is cølled after a GE f TATE, abl ø
tracing.)

(PRoG (xpos IDIFFERENCE LAsTNOUsEZ øTRACEWINnoNł øRIGNTTRACE/OFF
[ypoo IDIFFERENCE LAsTNOUsEY øTRACE*INr~ł øNEIGnTTRACE/OfFSsEETiJ)))
ø If there Is an ørror In thø function, ress thø riKbt
button to unodvlsø thø function. This øill eep thø ac inø
fr: locking up.)

(If (LASTNOUSESTATE RIGiiT)

(if ~t1h~~~ (NUNAPuISE (QUOTE 6ElS~5ESlATE)))
Q xpoa øLAST-TRACE-XP0sø

(KEO ,pea øLAsl-TeACE-YPOSøj)
thøn

ø Restoro ahøt ~s undør the eld pooltla of thø trøcø
OilUp)

(SITGLY øOLliIllnApposITIGøø o o (IREE5IlliA~)

ø2A SIMPLE lNTE~CJ\OHS WITH THE CURSOR. A BtTMAP. AflO A WINDOW
1

F,
FUNCTIONS FOR "TRACING THE CURSOR"

IPLUS CAR øil

IPLUS CADR ø:DDRRESEGISIrnø2øLASTTRAC

ø søvø øb&t ø111 bø undør thø position of thø nee trøcø
biteap)

(51 TILT SCREENBITNAP)
[IPLUS (CAR øaDREGIaø)
xpos)

(IPLUS øvNDREGIoaø O O)

BIfflLT the tracG blt:p onto thø nøø position of thø
eouse.)

(8ITBLT øTRACEBITNAPł O O ~5CREENBITNAP)
(IPLUS (CAR øilDRE ION")

(fPLUS (CADS øffORE6IONø)
ypos

NIL NIL YE INPUT)

(ONDTE P liT))

ø Savu the current position as the last trace position.)

(SETQ øLAST-TRACE-xPDSø xpos
(SETQ "LAST-TRACE-YPOSł ypos

The helper function for UHTRACE/CURSOR, called
UNDO/TRACE/DATA, must also be added to the environment:
(DEFINEQ (UNDo/TRACE/DATA
[LISA NIL

ø The purpose of this function is to turn tracing off and
to free up the global variables used to trace the bitaap, so
that thej can be garbage collected.)

Check if the cursor is currently being traced.
It so, turn it off.)

UiTRACE/CURSoR)

WINDoNPRDP łTliCE*IN~ł(uTE CURSDRINFN)
NIL)

(WINDo*PRDP łTRACEwINDDNł(uTE CUR~R0UTFN)
NIL)

SETQ "TRACEBITsAPł NIL)
SETQ øRIGNTTlicE/oFFsETø NIL)
SETQ øHEIGHTTRACE/OFFSETø NIL)
SETQ øOLDBITliPP0SITIDNø NIL)
SETQ łTRACE*I~ł NIL)

ø Turn GCGAG on)

(6C6A6 TJ))

Finally, if you inCluded the WlNDOWPROP to allow the user to place the bitmap in the window by pressing a mouse button, you must also type this function:

(D[E~DEAQ, ønd)
UNADVISE (SETNDUSESTATE))

fBITBLT øTliCEBITNAPø O O SCREENBIlNAP)
(IPLUS (CA 0Nø)
xpo

(IPLUS (CADR øiDREGIONø)
ypos)

NIL NIL (UTE INPUT)
(ATE PAINT]

That’s all the functions!

SIMPLE INTERAcTioNs NITH THE cuRsoR, A BITMAP, AND A WIND0W 42 S

p

RUNNING THE FuNcTlGh5

42.5 Running the Functions

To run the functions you just typed in, first set a variable to a window by typing something like
(SETQ EXMPLE.wIN~ (CRžATEI))

into the Jnterlisp-D Executive window, and sweeping out a new window. No’rv, set a variable to a bitmap, by typing, perhaps,
(SETQ ExlPLE.BTn (žDITl))
Type

(Estab1isfl’Trsce’Oo~ EXlPLE.WIN~ EXlPLE.BTK))
When you move the cursor into the window, the cursor will drag the bitmap.

(Note: If you want to be able to make menu selections while tracing the cursor, make sure that the hotspot of the cursor is set to the extreme right of the bitmap. Otherwise, the menu will be destroyed by the BITSLTs of the trace functions.)
To stop tracing, either

ø move the mouse cursor out of the window;
ø press the right mouse button;
ø call the function UNTRACE/CURS0R.

----- Next Message -----

Date: 19 Dec 91 17:30 PST
From: sybalsky:PARC:Xerox
To: sybalsky
Message-ID: <<91Dec19.173105pst.43009@origami.parc.xerox.com>.?::>

<----RFC822 headers-----
Received: from origami.parc.xerox.com ([13.1.100.224]) by alpha.xerox.com with SMTP id <11702>; Thu, 19 Dec 1991 17:31:18 PST
Received: by origami.parc.xerox.com id <43009>; Thu, 19 Dec 1991 17:31:05 -0800
From: John Sybalsky <sybalsky.PARC@xerox.com>
-----RFC822 headers---->

r fi. OTMER RžFERENCES THAT WILL
8E USEFUL TO YOU

Here are some references to works that will be useful to you in addition to this primer. Some of these you have already been referred to, such as:

ø The InterlispøD Reference Manual
ø The Library Packages Manual
ø The User’s Guide to SKETCH
ø Thell86orllO8User’sGuide

In addition, you can learn more about LISP with the books:
ø Interlisp-D: The languago and its usage by Steven H. Kaisler. This book was published in 1986 by John Wiley and Sons, NY.

ø Essential LISP by John Anderson, Albert Corbett, and Brian Reiser. This book was published in 1986 by Addison Wesley Publishing Company, Reading, MA. It was informed by research on how beginners learn LISP.

ø The Little Lisper by Daniel P. Friedman and Matthias Felleisen. The second edition of this book was published in 1986 by SRA Associates, Chicago. This book is a deceptively simple introduction to recursive programming and the flexible data structures provided by LISP.

ø LISP by Patrick Winston and Berthold Horn. The second edition of this book was published in 1985 by the Addison Wesley Publishing Company, Reading, MA.

ø LISP: A Gentle Introduction to Syabolic Coaputation by David S. Touretzky. This book was published in 1984 by the Harper and Row Publishing Company, NY.

Finally, there are three articles about the Interlisp Programming environment:

ø Poaer Tools Tor PrograffaersbyBeauSheil. It appeared in Datamation in February, 1983, Pages 131 - 144.
ø The Interlisp Prograffaing Environaent by Warren Teitelman and Larry Masinter. It appeared in April, 1981, in lEEE Computer, Volume 14:1, Pages 25 - 34.

ø Prograaøing In an Interactive Environaentø the LISP Experience by Erik Sandewall. It appeared in March, 1978, in the ACM Computing Surveys, Volume 10:1, pages 35 - 71.

Each of these articles was reprinted in the book Interactive
Prog raøøl ng Envl ronaents by David R. Barstow, Howard E.

Shrobe, and Erik Sandewail. This book was published in 1984 by
McGraw Hill, NY. The first article can be found on pages 19 - 30,
the second on pages 83 - 96, and the third on pages 31 80.

J:’

I

----- End Forwarded Messages -----
—End of message—

Venue

Medley for the Sun Workstation® User's Guide

Release 2.0
March, 1991

Venue

Medley for the Sun Workstation® User's Guide

Release 2.0
June, 1991

Address comments to:
Venue
User Documentation
1549 Industrial Road
San Carlos, CA 94070
415-508-9672

MEDLEY FOR THE SUN WORKSTATION® USER'S GUIDE

Release 2.0

June, 1991

Text was written and produced with Venue text formatting tools; Xerox printers were used to produce text masters. The typeface is Classic.

1. INTRODUCTION

Medley is an integrated programming environment, with support for the Interlisp and Common Lisp languages, an integrated windowing system, and a large collection of utilities and programs. It offers a mature and rich programming and development environment, as well as access to a large number of applications written for Interlisp, Interlisp-D, Common Lisp, and LOOPS.

Medley for the Sun Workstation has two versions, a Sun–3 version and a Sun–4 version, available on separate tapes. Medley 2.0 runs on the Sun–3 and Sun–4 workstations and the SPARCstation.

What Medley Requires

Hardware

Medley runs on Sun–3 and Sun–4 Workstations and the SPARCstation. It runs on both standalone workstations and diskless workstations linked to servers.

Medley on the Sun–3 Workstation requires the MC68881 floating-point coprocessor chip. On the Sun–4 Workstation, the Weitek 1164/1165 coprocessor is optional, but recommended.

For adequate performance, we recommend at least a 20 MHz 68020 (Sun 3/60 or 3/260), a 14 MHz SPARC (Sun 4/110 or 4/260), or a SPARCstation.

Except under X Windows, reasonable interactive performance can be expected with 8 megabytes (MB) or more of RAM. Smaller configurations of diskless workstations have been tested, but performance suffers. When using X Windows software, allow an additional 4 MB.

Naturally, larger applications will benefit from more memory. Medley's maximum working set is approximately 40 MB.

Input/Output Devices

Medley provides access to the Sun's input/output devices, such as display, keyboard, mouse, and file systems. It also provides access to PUP and XNS Ethernet services directly.

Bitmap Display

Medley supports all standard Sun displays and frame buffers.

Printers

You can print on Xerox Interpress printers using the XNS networking protocols. The FX80 printer also works via the RS232 port.

If you have a PostScript printer, you can use the LispUser modules PostScriptStream and UNIXPrint to direct output to your printer.

Software Requirements

Medley on the Sun–3 Workstation requires SunOS versions 3.2, 3.4, 3.5, 4.0, or 4.0.3. On the Sun–4 Workstation, Medley requires SunOS version 4.0, 4.0.3, or 4.1.

If you plan to run Medley under X Windows, you will need X11, version 4, or Motif.

NOTE: Medley's XNS Ethernet code will not work if you are running SunOS 3.5 configured for Kernel XNS Ethernet Support or Alpine.

Medley and Other Applications

Display Usage

When Medley is running alone, it takes over the entire display screen. When running under X, Medley uses one window as its screen; Medley maintains its own windows within that single window. Medley cannot run at the same time as Suntools or Open Windows.

Processor Usage

Medley runs its own process scheduler; as far as the UNIX scheduler is concerned, Medley is always running. For this reason, other heavy computational jobs on the same Sun Workstation will not get as good performance as they would competing with conventional UNIX interactive applications.

Similarly, Medley may not have adequate interactive performance if it is competing with other compute-bound tasks on the same machine.

For these reasons, we recommend that Medley be used on machines that are set up primarily for a single user.

System Components

Functionally, Medley consists of the following components:

emulator — A SunOS-executable program, which performs several functions. It executes the Interlisp-D virtual machine instruction set compatibly with the microcode of the Xerox 1100 series workstations. (This instruction set allows memory-efficient representation of Interlisp and Common Lisp programs.) It also provides access to the host machine's I/O (display, keyboard, file system), and executes some system functions directly.

sysout — A virtual memory image (the *sysout*) containing both byte-code-compiled Lisp functions and data structures. The sysout provided can be used both on the Sun Workstation and on the Xerox 1100 series machines.

library — Files of compiled Lisp code and data structures.

fonts — Data describing the "looks" of printed characters used by Medley's graphics, windowing, and hardcopying subsystems. Font directories are in three groups: display fonts, InterPress printer fonts, and Press printer fonts.

checksum — A script that reports inconsistent files, the correct checksum values for the files, and an error message. The checksum of individual files can be generated with the UNIX command `sum` *filename*. Use this when Medley installs correctly but does not run.

Medley Device-Naming Conventions

Medley for the Sun Workstation lets you interact with SunOS file systems (including file systems mounted from other machines) by using host device names. The two device names are as follows:

{`DSK`} A host name which gives you access to the SunOS file system using Xerox workstation local disk conventions.

{`UNIX`} A host name which gives you access to the file system using normal SunOS conventions.

The {`DSK`} device name provides an interface to the Sun Workstation for users who want to maintain compatibility with existing development tools and applications originally developed on a Xerox workstation. The {`UNIX`} device name provides a way for new applications to interact naturally with UNIX. Chapter 5 explains, in greater detail, some important exceptions and restrictions to the {`DSK`} and {`UNIX`} device name.

Notation Conventions

Text marked by a revision bar in the right margin contains information that was added or modified since the last release. Fonts, packages, and prompts have the following types of notation.

Fonts

`Bold text in TITAN font` indicates text you should type in exactly as printed.

`Regular TITAN` font text indicates what the system prints on your workstation screen. Lisp functions and variables and UNIX files and programs are also shown in `TITAN FONT`.

Text in Classic italics indicates variables or parameters that you should replace with the appropriate word or string.

Packages

Most Lisp symbols have a Lisp package qualifier; the `INTERLISP` package (`IL:`) is the default when no package qualifier is shown.

Prompts

All examples which include SunOS dialogues use the following conventions for the SunOS prompt.

A number sign (#), part of the system prompt, indicates that you are logged on as `root` or is running `su`; for example,

```
prompt#
```

A percent sign (`%`), part of the system prompt, indicates that a user other than `root` is logged on; for example,

```
prompt%
```

Compatibility

The Medley release on the Sun Workstation is designed for maximum compatibility with the Xerox workstation implementations. However, when moving applications to the Sun Workstation note the differences in end-of-line conventions and techniques for moving files.

Sysout Compatibility

Sysouts of the same version are compatible with all machine types. But a sysout generated on a Sun Workstation cannot be used on a Xerox workstation.

NOTE: You cannot mix different versions of sysouts and emulators.

Compiled-File Compatibility

Code compiled in a Medley 1.0, 1.1, 1.15 or 1.2 sysout cannot be loaded into Medley 2.0 sysouts, nor can code compiled in Medley 2.0 be loaded onto earlier sysouts. Code compiled for Medley 2.0 on a Xerox workstation cannot be loaded into Medley running on a Sun. The opposite is not possible either.

End-of-Line Convention

Some care must be taken in moving files to and from Xerox workstations, since the default end-of-line convention in UNIX is to terminate lines with the line feed (LF) character, while, traditionally, Medley has terminated lines with the carriage return (CR) character. In particular, if you use some other file transfer mechanism, such as FTP or Kermit, be careful to transfer `.TEDIT`, `.DFASL`, and `.LCOM` files in binary mode.

In Medley on the Sun Workstation, the default end-of-line convention for all text files is line feed (LF). The default end-of-line convention for all binary files is carriage return (CR); this is because CR (ASCII 13) is used internally in the system.

Release Contents

The release distribution contains the following documentation and software.

Documentation

The Medley documentation kit for users moving from a Xerox workstation to a Sun Workstation contains:

- *Lisp Library Modules*, Medley Release
- *Lisp Release Notes*, Medley Release
- *Medley For the Sun Workstation® User's Guide*

- Sun Type 3 and Type 4 keyboard templates.

New customers also receive the following:

- *Interlisp-D Reference Manual*, *Volumes 1-3*, Koto Release
- *Xerox Common Lisp Implementation Notes*, Lyric Release
- *Lisp Documentation Tools*, Lyric Release
- Guy Steele, *Common Lisp, the Language, First Edition*

All users can also purchase this document:

- *LispUsers' Modules*, Medley Release

Software

The software release is available on either a ¼-inch tape cartridge or a ½-inch 9-track tape. The software release is specific to the Sun architecture (Sun 3 or 4) for which you purchased Medley, but contains multiple SunOS versions. This `tar` tape contains the directories listed below. (See Appendix C for details of the directory contents.)

```
./install-medley
./medley
./install.sunos3/
./install.sunos4/
./install.sunos4.1/
./lisplibrary
./checksumdir
./lispsysouts
./fonts/display
./fonts/interpress
```

LispUsers Modules

The Medley version of LispUsers Modules is a software supplement to Medley for the Sun Workstation. This is software written by our users which you may purchase separately. The support for these modules comes from each module's author; Venue has no commitment to support LispUsers' modules.

Two LispUsers Modules are particularly useful when you are running Medley on a Sun Workstation. For those users with Postscriptstream printers for output, the PostScript module is particularly useful. The LispUsers module RPC implements Sun remote procedure calls.

[This page intentionally left blank]

TABLE of CONTENTS

[This page intentionally left blank]

2. SOFTWARE INSTALLATION

This chapter describes how to install Medley on Sun Workstations.

To install the Medley Release on a Sun Workstation, you need the following:

- Release tape
- Medley documentation kit for the Sun Workstation.

Getting Ready to Install Medley

Medley includes a shell script for automatic installation. The script infers as much as it can about your host and network, but will prompt you for answers when needed. Once it has collected the necessary information, it installs only those parts of Medley that you really need.

Do not worry if you forget something. You can run the installation again, and pick up any items you missed.

Before installing Medley, you should gather some facts about the hardware and network environment on which you will be using Medley. The following checklist will help you.

- Do you have the correct release tape correct for the kind of machine on which you plan to run?

  The tape is labeled either Sun-3 or Sun-4/SPARCstation.

- Where is the tape drive you will be using?

  Does your Sun have a ¼-inch tape drive? If not, you need a Sun with a ¼-inch tape drive on your network. You will need to know the host name for that machine.

- Does your system have sufficient swap space (45 MB) for Medley?

  If you are not sure, see the section below for instructions.

- Are you installing Medley for a single user, or will several users be sharing this copy?

  If it is for shared use, you will want to install Medley on a public directory on a shared server. For that, we recommend using `/usr/share/lde` as the directory name. You will need to be running on the server when you do the installation, and you will probably need to log in as `root`. Check with your System Administrator for details.

- Do you have enough disk space free ?

  You need to select a file system with enough disk space to install the software. A minimal installation requires approximatelly 12 MB, and a full installation will require up to 23 MB. Use the UNIX command `df` to find one.

- Have you selected an installation directory?

  The directory must be on a file system with enough space. For individual use, we recommend `/usr/local/lde`.

CAUTION

If the installation directory contains a previously installed version of Medley, some of the older files will be replaced with new ones.

- Do you have write permission to create the installation directory and to write files into it?
- Will you be running on X Windows?

  If you are, you will need X11 R4 or Motif. If you have X Windows installed, the utility will install the software necessary to run Medley under X11.
- Will you be running XNS (Xerox Network Services) or PUP protocols?
  - If so, you must be logged in as `root` during the installation.
  - If so, be sure you are not running SunOS 3.5 Kernel XNS Ethernet Code ("Alpine Kernel").
- If your host is networked, do you have XNS (Xerox Network Services) servers on it?

  If so, you will want to install XNS-relevant software. Also, if you have XNS Print Services and InterPress printers on your network, you might want to install InterPress fonts, allowing you to use an InterPress printer from within Medley. The default is not to install any XNS-relevant software.
- Is this a new installation, or are you upgrading from an earlier version of Medley?

  If you are upgrading, you only need to install the sysout, the appropriate emulator, and library files. If you are making a new installation, you will need at least the display fonts as well.

Ensuring Adequate Swap Space

Medley requires 45 MB of swap space on top of the normal swap space requirements. Check swap space using the `pstat` command:

```
prompt%  /etc/pstat -s
37176k used (3176k text), 12920k free, 1344k wasted, 0k missing
max process allocable = 10224k
avail: 5*2048k 1*512k 4*256k 3*128k 6*64k 7*32k 7*16k 40*1k
4800k allocated + 2520k reserved = 7320k used, 64672k available
```

If you need more swap space, consult the *Sun Software Technical Bulletin, March 1988*, pages 335-36, for information on increasing the amount of available swap space.

Installing Medley for Shared Use

If several people will be be using Medley on different machines, it probably makes sense to install one copy and have people share it.

You will need to find one machine—probably your main file server—with enough disk space. You will also want to have the directory look the same to every user. We recommend calling the installation directory `/usr/share/lde`.

Installing Medley Software

1. Log in under your username.

   login *yourname*

2. Put the tape in the tape drive. The script will allow you to install from a tape drive on a remote host. If you are performing a remote installation, put the tape in the tape drive of the host.

3. Retrieve the installation utility from the tape, as follows.

   - If the tape drive is on a different host, enter the following:

     rsh *remote-host* **dd if=/dev/rst0 | tar xf -**

     Replace *remote-host* with the name of the host on your network that has the tape drive you are using. This copies the file `install-medley` to your working directory.

   - If the tape drive is local to your machine, type the following:

     tar xf /dev/rst0

     This copies the file `install-medley` to your working directory.

   If you have any problems during this step of the installation, consult your local UNIX system administrator.

4. Run the installation utility:

   install-medley

   Each time you are prompted for information, the script will show you the default in [ ] brackets. Pressing the return key selects the default. To select a different option, type it.

   After installing Medley, the script will offer to update two files which must refer to the installation directory. We recommend this. The two files are:

   medley A script for running Medley easily

   site-init A sample site-init file

You can stop the installation process at any time by typing ↑**C** (Control-C).

Using the Installation Script's Menu

To choose an option from the Installation Options Menu, type at least the first three characters of the selection. Most of the menu lists items you might want to install. Choosing one of these options works as a toggle switch, either selecting or deselecting, depending on its previous setting. The other menu items act as commands when you select them. "`OS version`" lets you pick the SunOS version(s) for which you will need matching emulators. "`Directory`" lets you specify where to install Medley. "`!`" lets you use UNIX commands if you need to.

In Figure 2-1, the user has selected for installation the Sysout, Monochrome and X Windows emulators for SunOS 4.1, Display fonts, and Library modules. The menu shows that you need 15052.8 KB of disk space to finish the installation, but only 13002 KB are available. At this point you can either deselect an option to decrease the disk space requirements, or change the installation directory to one that has sufficient disk space.

```
<----------------> Installation Options Menu <---------------->
------------------------ Emulators -------------------------
For one or several OS versions (At least one of monochrome,
color or X11-version is required for new installations)
   x  Monochrome - 0.5 MByte
   -  Color
   x  X11-version - 0.6 MByte
   -  XNS - allows use of XNS protocols
   -  Object files - allows linking of Medley to other software
      OS version - Change versions. Selected: 4.1
-------------------------- Fonts ------------------------------
   x  Display - 5.5 MByte (recommended)
   -  Interpress
----------- Sysout, Library & Checksum files --------------------
   x  Sysout - 5.1 MByte (required for new installations)
   x  Library modules - 3 MByte (recommended)
   -  Checksum files
------------------------ Commands ------------------------------
      Directory - Change location of installation directory.
        -- Current: /usr/share/lde 13002 KB
        -- Disk-space(KByte) Available:13002 Needed:15052.8
      !<Unix command> - Execute a Unix command
      ? or Help - Show menu instructions
      Redraw - Redisplay this menu
      None - Unmark all options
      All - Mark all options
      Continue installation
      Quit installation
Select [Directory]:
```

Figure 2-1. Sample Installation Menu

Because of the disk space shortage, the script has offered `[Directory]` as the default next command. If that is what you want, just press Return. Otherwise, type some other command.

For example, to deselect `Library modules`, type:

```
Select [Directory]:lib
```

Alternately, to find a filesystem with enough disk space, issue the following command:

```
Select [Directory]: !df
```

```
Filesystem        kbytes      used     avail  capacity Mounted on
/dev/sd0a           7548      4935      1858      73%    /
/dev/sd0g          75106     51704     15891      76%    /usr
/dev/sd1g          47999     21898     21301      51%    /home
king:/shared      416928    349202     26042      93%    /share
Select [Directory]:
```

To change the installation directory, type:

```
Select [Directory]:dir
```

At the new prompt, give the directory name:

```
Where do you want to install Medley? [/usr/share/lde]: /share/lde
```

If the directory does not exist, the script will attempt to create it.

Getting a Copy Protection Key

Before starting Medley, you must first obtain a host access key from Venue. This key enables the use of Medley on one workstation. The software cannot run without the key. You must have one key for each host on which you wish to run. Note that your current host access key *will* work if you move from one version of SunOS to another, or if you upgrade from an earlier version of Medley to Medley 2.0.

To obtain a key:

1. Get the host ID of the machine on which you intend to run Medley by typing on that machine

   ```
   prompt% hostid
   310002f6
   ```

2. Call Venue at 800-228-5325 between 9:00 a.m. and 4:30 p.m. PST. Outside the United States, call your local distributor.
3. Ask for a host access key, giving the Venue representative your host ID. Venue provides you with a host access key, which you need during software configuration, below.
4. If you plan to use the Medley startup script, you will be prompted for the key the first time you invoke it. The script will automatically save the key into a file for future reference.

CAUTION

Depending on your license agreement with Venue, your host access key may have an expiration date. After that date, your key is no longer valid.

Changing Configurations or Adding Options

If you forgot to install something or need to add a new emulator, you can rerun the installation, and select only the new things you need. The installation script does not remove things.

This need arises most often when you start running X Windows and need the new emulator.

First, decide what you need to install. Then type **install-medley**. When you get to the Installation Options Menu, select all the new things you need and *deselect* everything else. Continuing the installation from there just adds the new items.

If you have Sun-3s and you just got a SPARCstation, all you need from the new installation are the proper emulators. Everything else is the same.

[This page intentionally left blank]

LIST of FIGURES
LIST of TABLES

[This page intentionally left blank]

3. GETTING STARTED

Getting Ready

To prepare your system to run Medley, use the following checklist:

1. Exit `suntools` or any other windowing system, unless you plan to run under X (in which case you can leave X running). Medley provides its own window system and must not run simultaneously with others.
2. Kill all your user processes (these have console as the control tty). Check to make sure you have killed any "`selection_svc`" process. If you do not perform this step, certain error messages from UNIX (e.g., `file system full`) cause those processes to print to the console, resulting in scrolling of the display.
3. Check for the directory for the software, and add it to your path, if necessary:

   `prompt#` **`set path = ($path /usr/share/lde/install.sunos`***`x`***`)`**

   You can also add this to your `.login` file.

Running Medley

Running Medley Directly

Invoke Medley by typing the name of the program, e.g.,

`prompt%` **`lde`** *optional-sysout* **`-k '`***thishost-key***`'`** `[-m` *memory-size*`]`

If you are using either Xerox XNS or Xerox PUP Ethernet protocols, type instead

`prompt%` **`ldeether`** *optional-sysout* **`-k '`***thishost-key***`'`** `[-m` *memory-size*`]`

This, in turn, runs **`lde`** and lets it use the Ethernet directly.

optional-sysout is the name of a Lisp virtual memory image file (see the section Where Medley Looks for Your Sysout below). *thishost-key* is the key you obtained from Venue for the machine on which you are running.

If the sysout was created on a machine with a different size display, the image will appear garbled for several seconds. After Lisp starts running, it readjusts the display to the current size.

The `-m` flag lets you control the maximum amount of memory Medley will use. *memory-size* is a number in the range 8 through 32, in megabytes. (See the detailed explanation on page 25.)

Using the Medley Shell Script

The script will try to find a key, an appropriate emulator, and a sysout. The script relies on information about where the Medley software was initially installed on your system. (The installation script `install-medley` automatically updates this information for you.) The `medley` script assumes that you have not changed the installation subdirectory structure from when it was originally installed.

The script will first try to find a key in the file *Installation directory*/ `.medleyKey.`*hostname* or in `~/.medleyKey.`*hostname*. *Installation directory* is where Medley was originally installed on your system. *hostname* is the name of the host for which the license key was issued. If neither file is found, you will be prompted for a key.

`medley [[`*emulator*`]` *sysout*`]` [Command]

emulator Given a pathname or a simple name, the command will search for *emulator* as follows:

- If *emulator* is a relative or absolute pathname, e.g., `/share/medley/emulators/lde`, it will only try that pathname.
- If *emulator* is a plain file name, e.g., `lde`, the script uses the regular UNIX search path to find it. If it cannot find it, the script looks in the installation directory for Medley at your site, e.g., `/usr/share/lde/install.sunos4.1/lde`.
- If you omit *emulator*, the script uses `lde` as the default value, searching for it in the same fashion as above.

sysout The command will search for *sysout* as follows:

- If *sysout* is a relative or absolute pathname, e.g., `../applications/my.sysout`, it will only try that pathname.
- If *emulator* is a plain file name, e.g., `my.sysout`, it will look for it in the following order:
  1. Current working directory `my.sysout`
  2. Installation directory for Medley at your site, e.g., `/usr/share/lde/lispsysouts/my.sysout`.
  3. Your home directory, `~/my.sysout`
  4. The `medley` subdirectory in your home directory, `~/medley/my.sysout`.
- If you omit *sysout*, the script looks for it as explained in the Where Medley Looks for Your Sysout section below.

Examples

- `prompt%` **`medley`**

```
To start Medley 2.0, a host access key is required.
Call Venue at (1-800-228-5325) for one,
and be prepared to give them your workstations host ID#
Your workstations host ID# is: 51006da3
Type in key or [^C] to abort:8bf7723e 459aab34 73491feb
Saving key '8bf7723e 459aab34 73491feb' into file
'.medleyKey.hostname'...
Trying /usr/share/lde/.medleyKey.hostname ... Write
protected !
```

```
Trying home-directory/.medleyKey.hostname ... Done
Starting up Medley 2.0 ...
.................... Medley 2.0 starts ....................
```

If you had Medley installed in `/share/medley` on your system, it would try to run the emulator `/share/medley/install.sunos4.1/lde`, using the sysout `/share/medley/lispsysouts/LISP.SYSOUT`.

In this example you are prompted for a key, which is saved into the file: *home-directory*`/.medleyKey.`*hostname*

The script tried to save the key into the installation directory but did not have write access there. Instead it was put into your home directory (~/) . *hostname* is the name of the host running `medley`.

The next time you use the script `medley`, you will not be prompted for the key.

- `prompt%` **`medley application.sysout`**

  If you had `application.sysout` in your home directory, it would try running the emulator `/share/medley/install.sunos4.1/lde` using `~/application.sysout`.

Where Medley Looks for Your Sysout

If you run Medley directly, the system searches the following places, in order, for the sysout to be used:

- *command line*

  The name of the sysout file can be given on the command line when starting Medley; for example,

  `prompt%` **`lde`** *sysout* **`-k`** **'***thishost-key***'**

- `LDESRCESYSOUT`

  If no sysout file name is given on the command line, the value of the environment variable `LDESRCESYSOUT` is used as the name of the sysout file. For example:

  `prompt%` **`setenv`** `LDESRCESYSOUT` *my.sysout*

  `prompt%` **`lde -k`** **'***thishost-key***'**

  would run the host key *my.sysout*.

- `~/lisp.virtualmem`

  Finally, Medley looks for the file `lisp.virtualmem` on your home directory.

Where Medley Looks for Your Site Initialization File

When Medley starts, it reads in a Lisp site initialization file. This site initialization file sets things like pathnames for fonts, site parameters, and the like.

Greeting and initialization are described in the *Interlisp-D Reference Manual*, Section 12.1.

Medley looks for a site initialization file in a number of locations:

- `LDEINIT`

  If the environment variable `LDEINIT` is set to a complete Lisp file name, Lisp looks there first for the site initialization file:

  ```
  prompt% setenv LDEINIT /usr/lisp/my-site-init.lisp
  ```

- `/usr/share/lde/site-init.lisp`

  If `LDEINIT` is not set or there is no file with the name given, Lisp looks for a site initialization file called `/usr/share/lde/site-init.lisp`. The distribution tape contains a sample site initialization file in the Lisp library directory `/usr/share/lde/lisplibrary/site-init`. The system administrator should copy `site-init` into `/usr/share/lde/site-init.lisp` then customize it for the site. The comments in the sample `site-init.lisp` describe the parameters it sets and give guidelines for customizing it to your local conditions.

- `{DSK}INIT.DFASL`, `{DSK}INIT.LCOM`, `{DSK}INIT.LISP`

  Finally, Lisp looks for a site initialization file on your Medley home directory (`{DSK}`). Chapter 5, Medley File Systems, describes the `{DSK}` device.

Medley and X Windows

Medley 2.0-S supports the X Window System, Version 11 Release 4 (X11R4). Medley runs in a single X window; Medley's "screen" is displayed in that window, and you use Medley as usual.

Starting X Windows

Start the X server on your console. Use the `xinit` command.

If necessary, start a window manager as a client of X (`xinit` often starts a window manager). The window manager provides many window management functions, such as moving, resizing and iconifying the window. Medley has no window management function of its own.

Running Medley Remotely

You can run Medley on one machine, with the window on some other machine. To do so, perform these steps on the machine whose keyboard and display you will be using:

1. Add the host name to execute the Medley access control list:

   `xhost +` *hostname*

2. Open a new xterm and rlogin to the Sun Workstation on which Medley is to run. Set the environment variable `DISPLAY` to the host name of the server machine:

   `setenv DISPLAY` *servername***`:0`**

3. Set the `LDEKBDTYPE` environment variable to tell Medley what kind of keyboard you will be using. Possible values are:

`type3`	Sun Type 3 keyboard
`type4`	Sun Type 4 keyboard
`rs6000`	IBM RS/6000 or PS/2
`dec3100`	DECstation 3100 or 5000

`hp9000`	HP9000 Series 700 or 800
`X`	Generic X terminal

If you don't set `LDEKBDTYPE`, it will default to `X`. The advantage of specifying a specific keyboard lies in how Medley treats the special function keys. The specific keyboard maps maximize the usefulness of keys marked, e.g., "Find". The generic keyboard code cannot do that reliably.

4. Start up Medley.

A new window for Medley will appear on the X server's screen.

The Medley Window

Normally, Medley uses the whose screen. Under X, Medley's "screen" appears in a single X window. Medley's screen is slightly smaller than the screen you are using to display it; if you make the X window full–screen–size, you see Medley's entire screen. If it is smaller, you will need to scroll to see parts of the screen.

The scroll bars (at the right and bottom of the X window) control what parts of Medley's screen appears in the window. Use the vertical scrollbar to scroll up and down, and the horizontal scrollbar to scroll left and right. The gravity buttons (at the lower right corner) set the bitgravity of the display window. Click the mouse button on one of these areas. The shade pattern is moved to the clicked area, and the bitgravity is set in the corresponding corner on the display window. The bitgravity determines how reshaping the X window affects what part of the Medley screen is visible.

Environment Variables

Medley on the Sun uses several environment variables. They can be set from the shell with the **setenv** UNIX command. By convention, environment variable names use uppercase rather than lowercase letters, e.g., `LDEDESTSYSOUT`. The Medley environment variables are listed below, with a reference to sections in this *Guide* where further information can be found.

`LDEKBDTYPE`	See the Medley and X Windows or Sun Type 4 Keyboard sections in this chapter.
`LDEINIT`	See the Site Initialization File section in Chapter 4.
`LDESRCESYSOUT`	See the Where Medley Looks for Your Sysout section in this chapter.
`LDEDESTSYSOUT`	See the Saving Your State section in Chapter 4.
`LDESHELL`	See the UNIXCHAT section of the *Lisp Library Modules*.
`LDEFILETIMEOUT`	See the File System Errors subsection in Chapter 5.

Keyboard Interpretation

This section describes how Medley interprets the Sun Type 3 and Type 4 keyboards. Except when running under X, Medley performs its own keyboard interpretation, taking raw up/down transitions directly from the keyboard. Medley uses its own key numbering scheme; key numbers are used by Lisp functions such as `IL:KEYDOWNP` and `IL:KEYACTION`.

These key assignments were chosen to maximize compatibility with both the Xerox workstation keyboard and the normal Sun keyboards. You can attach a Sun Type 3 or Type 4 keyboard template, which also shows the Medley keyboard assignments, to your Sun Type 3 or Type 4 keyboard. Both templates are included with your Medley documentation set.

Sun Type 3 Keyboard

Figure 3-1 shows the key number assignments for the Sun Type 3 keyboard. Figures 3 - 2 through 3 - 4 show Medley's key assignments for the Sun Type 3 keypads.

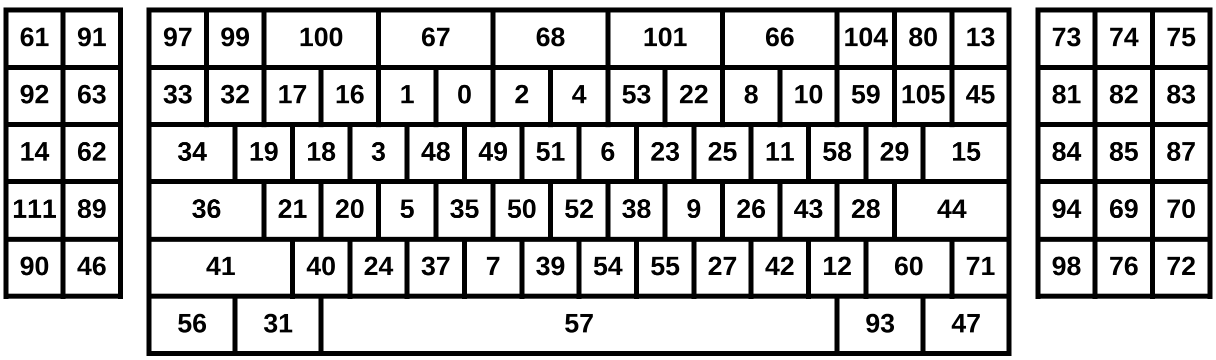


Figure 3-1. Sun Type 3 Key Numbering

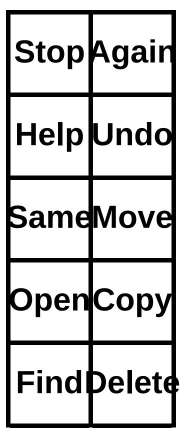


Figure 3-2. Sun Type 3 Left Key Pad

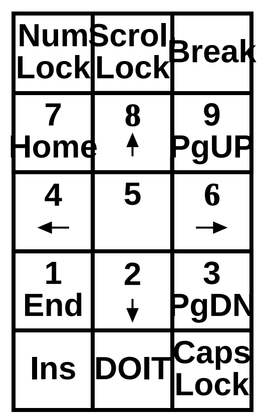


Figure 3-3. Sun Type 3 Right Key Pad

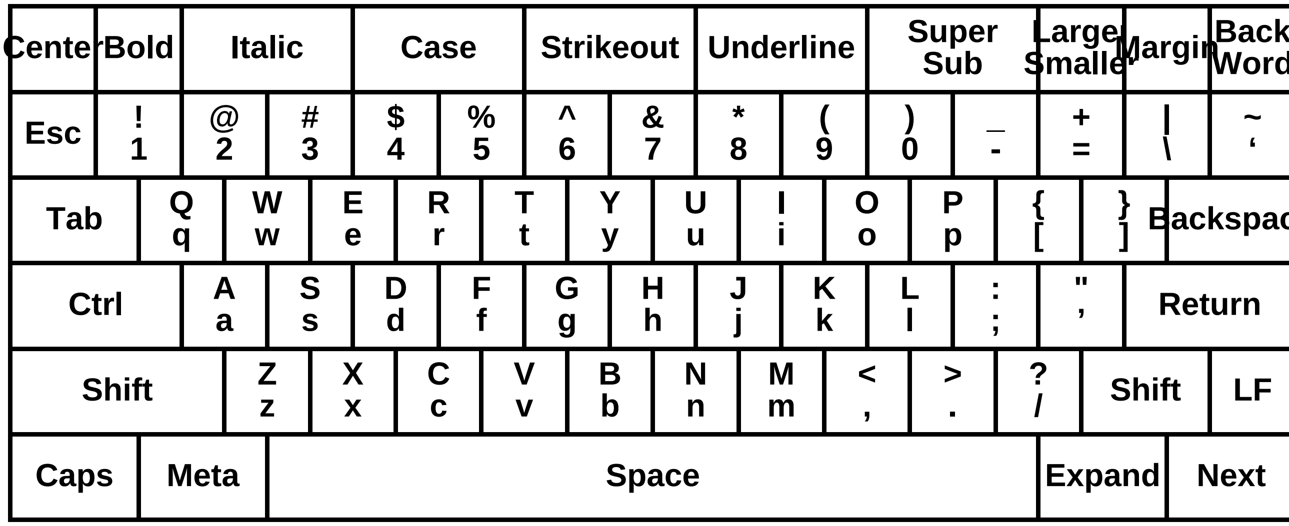


Figure 3-4. Sun Type 3 Center Key Pad Interpretation

Sun Type 4 Keyboard

Figure 3-5 illustrates the keyboard interpretation for the Sun Type 4 keyboard.Figures 3-6 through 3-8 show the keyboard and the left and right key pads for the Sun Type 4 keyboard.

NOTES: In SunOS 4.0, the NEXT (ALT/GRAPH) key on the Type 4 keyboard is inaccessible. Later versions of SunOS fix this.

Medley cannot detect whether it is running on a workstation with a Type 4 keyboard when running SunOS 4.0, 4.0.1, or 4.1. To make it work correctly on your workstation, enter the following before you start running Medley:

```
setenv LDEKBDTYPE type4
```

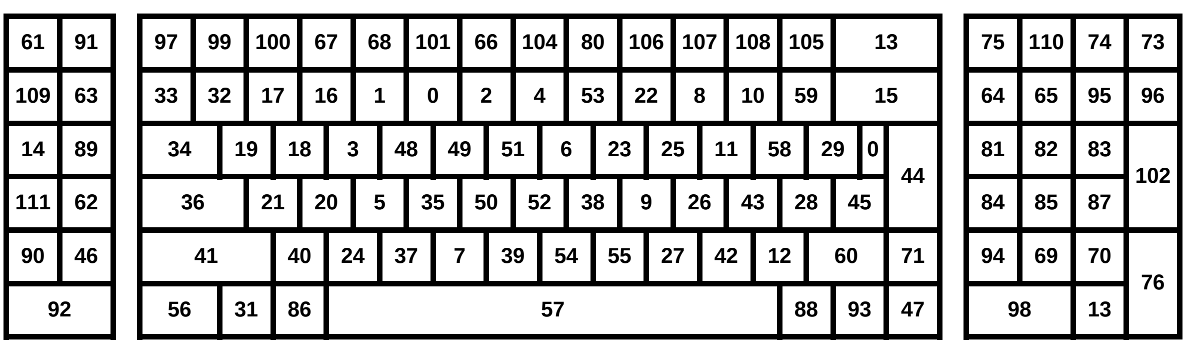
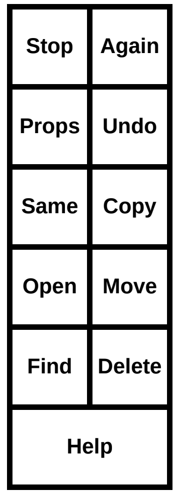


Figure 3-5. Sun Type 4 Key Numbering



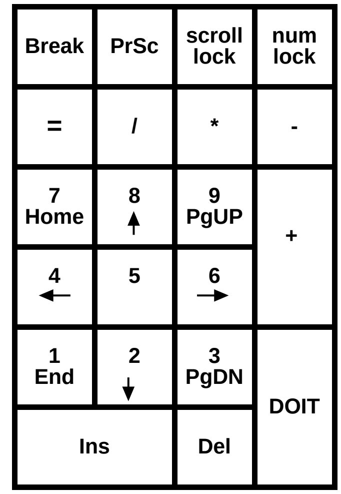


Figure 3-6. Sun Type 4 Left Key Pad

Figure 3-7. Sun Type 4 Right Key Pad

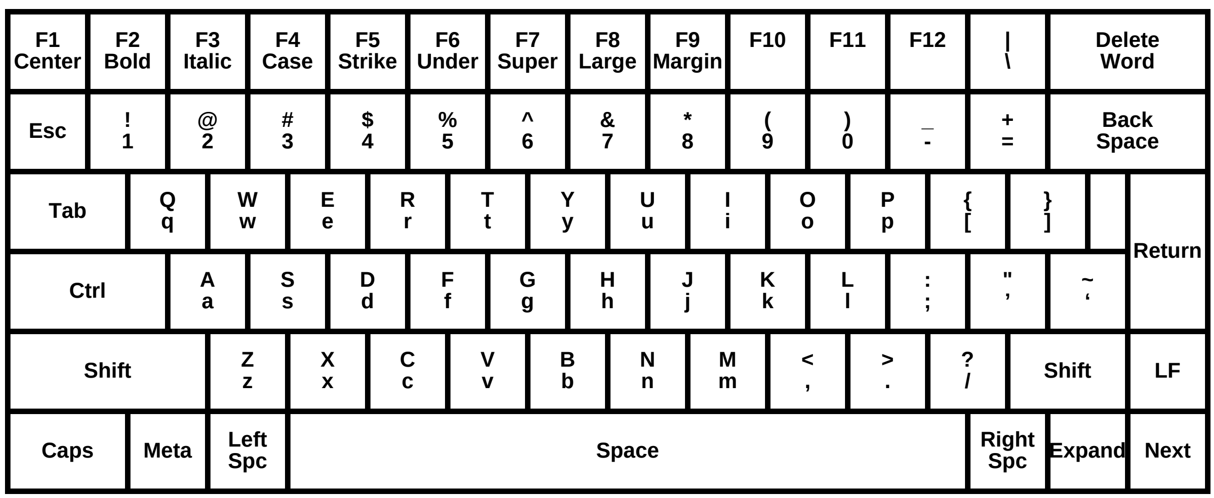


Figure 3-8. Sun Type 4 Center Key Pad Interpretation

[This page intentionally left blank]

PREFACE

This *Guide* describes Medley release 2.0 for the Sun–3 and Sun–4 workstations and the SPARCstation: the release contents, instructions for installing the release, and information on using it.

Audience

The *Medley For the Sun Workstation® User's Guide* is intended for users familiar with the Medley environment who want to use it on the Sun–3 or Sun–4 workstations or the SPARCstation. The *Guide* assumes that the user is already familiar with UNIX and SunOS concepts. The system administrator of a Sun system or network should read this *Guide* to ensure the correct installation of the Medley software.

Chapter 1 of this manual gives an overview of the product and its internal architecture, and is of interest to all users of the system.

System administrators should read Chapter 2, Software Installation; and Chapter 3, Getting Started. These chapters guide the administrator through the process of installing Medley 2.0 and configuring it on the Sun Workstation. Experienced Lisp users may want to configure the software; this procedure is described in Chapter 4.

Users already familiar with the Lisp environment on Xerox workstations should find Chapter 1 and Chapters 3 through 6 useful. These chapters describe the operation of the system after it has been installed as well as those functions and operations which are specific to the Sun Workstation.

Using This Manual

Chapter 1, Introduction, describes the hardware, input/output devices, and software needed to run Medley on a Sun Workstation; describes Medley and how it works with other applications; lists the system components; introduces pertinent SunOS and UNIX conventions used throughout the *Guide;* explains Medley's compatibility; and lists the contents of the release.

Chapter 2, Software Installation, contains the installation and software configuration procedures.

Chapter 3, Getting Started, explains how to set up a site initialization file and install the X Windows System. It also shows the keyboard configuration and has instructions for getting started in Lisp on the Sun Workstation.

Chapter 4, Using Medley on the Sun Workstation, describes how specific Lisp functionality works on the Sun.

Chapter 5, Medley File Systems, discusses the file conventions that need to be followed when running in Medley on a Sun Workstation. Differences in Lisp file attributes and variables are also discussed.

Chapter 6, Error Recovery, describes the diagnostic error recovery program URAID. This chapter explains how to recover from fatal error conditions and lists specific Lisp errors that may be encountered when running Medley on the Sun.

Appendix A, Installation Hints, contains additional notes to help configure Medley, and includes a complete description of the installation script.

Appendix B, Verifying the Installation Tape's Validity, tells how to validate the contents of the `tar` tape.

Appendix C, Layout of Installation Tape Files, includes a listing of the `tar` tape directories and the font directories.

Appendix D, Differences between Xerox Workstations and the UNIX Version of Medley, includes functions for controlling device-specific behavior of the Xerox 1100 series workstation disk drives. It also describes the library modules not supported on the Sun.

The **Glossary** provides definitions of SunOS, UNIX, and Lisp terms used in this *Guide*.

Medley is a Venue product which was built on the Xerox Lisp environment. It provides an integrated programming environment consisting of Interlisp-D and Common Lisp, a windowing system, and a set of programs and utilities. Users not already familiar with the Xerox Lisp environment should try to become somewhat familiar with it before attempting serious development work.

Supporting Documentation

The following reference documents are useful to have on hand during the installation process and when working in Medley on the Sun Workstation.

Sun References

This literature from the Sun documentation set is useful during the installation and when running Medley on a Sun Workstation.

- *Installing UNIX on the Sun Workstation*
- *UNIX Interface Reference Manual*
- *SunOS Reference Manual*
- *Sun Software Technical Bulletin, March 1988*

Venue Documentation

In addition to this *Guide*, the following documents describe the Medley system:

- Guy Steele, *Common Lisp, the Language, First Edition*, Bedford, MA: Digital Press, 1987
- *Interlisp-D Reference Manual, Volumes 1-3,* Koto Release
- *Xerox Common Lisp Implementation Notes*, Lyric Release
- *Lisp Documentation Tools*, Lyric Release
- *Lisp Library Modules*, Medley Release
- *Lisp Release Notes*, Medley Release

Templates for the Type 3 and Type 4 Sun keyboards are also part of the Medley documentation set.

New users of Medley receive, in the software kit, all the manuals listed above.

Users who are moving the Medley environment from a Xerox workstation to a Sun Workstation receive the following documentation in the software kit:

- *Lisp Release Notes*, Medley Release
- *Lisp Library Modules*, Medley Release
- *Medley for the Sun Workstation® User's Guide*
- Sun Type 3 and Type 4 keyboard templates

The manual *LispUsers' Modules*, Medley Release, which may be purchased separately, supplements the Medley release.

[This page intentionally left blank]

4. USING MEDLEY ON THE SUN WORKSTATION

Once the system administrator has installed Medley software on the Sun, Lisp users can customize their Medley Lisp environments. This chapter provides basic information to get you started in the Medley environment on a Sun Workstation.

Setting Up a Site Init File

The users at a given site generally print to the same printers, load library files from the same directory, and so on. Medley uses variables to supply defaults for such things. The obvious place to set these variables is in one common initialization file. That is the Site Init File's role.

The Site Init File is a file of Lisp expressions that is loaded when you start Medley with a fresh `LISP.SYSOUT`.

The following Lisp symbols should be set in your site init file:

IL:USERGREETFILES **[Variable]**

A list of templates to search for the place where individuals should find their personal init files. If this is not set in the site init file, no personal init file is used. The list should be similar to the following:

(({*file-server*}< USER >LISP>INIT.LCOM)
({*file-server*}< USER >LISP>INIT)
({*file-server*}< USER >INIT.LISP))

IL:DISPLAYFONTDIRECTORIES **[Variable]**

A list of directories to search when the system is looking for display fonts. The site initialization file should set it to a list of strings, each containing a complete pathname for font files, e.g., (`"{UNIX}/usr/local/lde/fonts/display/presentation/"`).

IL:INTERPRESSFONTDIRECTORIES **[Variable]**

A list of directories to search when the system is looking for Interpress font widths.

IL:DIRECTORIES **[Variable]**

The list of paths to search for files that are not found in the current (Lisp) connected directory.

IL:LISPUSERSDIRECTORIES **[Variable]**

The list of paths to search for library and LispUsers' files. Remember that every path in this list should also be in `DIRECTORIES`.

IL:DEFAULTPRINTINGHOST **[Variable]**

A list of names of default printers.

IL:DEFAULTPRINTERTYPE **[Variable]**

The default printer type, e.g., `POSTSCRIPT`.

XCL:*LONG-SITE-NAME* **[Variable]**

The value of the function XCL:LONG-SITE-NAME, e.g., "Frobnitz, Baz and Lispers, Incorporated."

XCL:*SHORT-SITE-NAME* **[Variable]**

The value of the Common Lisp function XCL:SHORT-SITE-NAME, e.g., "Frobco".

IL:\BeginDST **[Variable]**

The day of the year on or before which Daylight Savings Time takes effect (i.e., the Sunday on or immediately preceding this day). Must be set to 98 in the USA if Lisp is to perform time computations correctly (subject, of course, to future legislation). If you are in a region where Daylight Savings Time is not observed, set the value to 367.

IL:\EndDST **[Variable]**

The day of the year on or before which Daylight Savings Time ends. Must be set to 305 in the USA.

Setting Up a Personal Init File

Your personal init file keeps track of the location of your home directory and windows layout; it also remembers which library files you always load.

Your personal init file is a file of Lisp expressions that is loaded and run after the site init file. You can create it either as a text file, or have Medley's File Manager help you.

Your initialization file is normally ~/INIT.LCOM

Saving Your State

On the Sun, lde is an ordinary UNIX program that allocates a 45 MB data area, reads into that area several megabytes of data (the sysout), and modifies it there. Under UNIX, that program's data requirements (which include the sysout) are handled by UNIX; all Medley does is modify in "memory" a copy of your original sysout file. UNIX, transparently to Medley, handles all real memory swapping. This has several consequences related to starting, saving, and restarting sysouts.

On Xerox workstations, the virtual memory partition is updated periodically and used to store new pages as they are allocated or flushed from the real memory of the machine. For example, LOGOUT and SAVEVM write out only those pages of data which are different from what might already be in the virtual memory file.

On the Sun Workstation, however, the contents of virtual memory are only written to a file by an explicit call to SAVEVM, LOGOUT, SYSOUT, or MAKESYS. This file is an ordinary SunOS file (normally ~/lisp.virtualmem). The entire virtual memory, which may be many megabytes of data, is written out there.

On the Sun Workstation, starting anew from a saved virtual memory file requires reading it into memory. On the Xerox workstation, it is necessary to first copy the saved sysout to the virtual memory file and then read it in. Thus, restarting a saved sysout or virtual memory file is significantly faster on a Sun Workstation.

The file that LOGOUT and SAVEVM writes is normally ~/lisp.virtualmem (i.e., the file lisp.virtualmem on the user's home directory). However, the environment variable LDEDESTSYSOUT can be used to override this default. For example, you might want to keep virtual memory images on /user/local. During a demonstration where you do not want the memory image saved, you can reset LDEDESTSYSOUT to /dev/null. You can use the C-Shell command setenv to do this, e.g.:

prompt% setenv LDEDESTSYSOUT "/dev/null"

Cursor tracking interferes with writing out the screen bitmap as part of the Medley memory image. For this reason, Medley takes the cursor down before saving a virtual memory image as part of LOGOUT, SAVEVM, SYSOUT, or MAKESYS. When this happens, the message

```
Saving VMem, taking mouse down
```

appears in the prompt window, and cursor tracking is disabled.

Because the virtual memory file need not already exist to run Medley, the functions LOGOUT and SAVEVM can signal the following file errors:

```
File-System-Resources-Exceeded
Protection-Violation
File-Wont-Open
```

Even if some errors occur while saving a virtual memory, the old destination file is safe. Saving does not overwrite the old virtual memory file. The saving virtual memory file is named with "–temp", such as lisp.virtualmem–temp. The file is renamed to a specified name, such as lisp.virtualmem, at the last sequence of the save.

When the user does not have enough space to save the virtual memory, the old virtual memory file can be overwritten by setting IL:\LDEDESTOVERWRITE to T. The initial value of IL:\LDEDESTOVERWRITE is NIL. In some cases, even if the user tries to overwrite, there may still not be enough space.

In Medley, a "page" is 512 bytes. Under SunOS, the page size is variable; some Sun Workstations use 8 Kbyte pages. In general, Medley functions deal only in units of Medley pages, e.g., the SIZE attribute of files is in terms of 512–byte pages, (VMEMSIZE) returns the number of 512–byte pages in use.

(IL:LOGOUT ***FAST*****)** **[Function]**

Lets you exit Medley cleanly. The parameter *FAST* indicates whether resumption of the same environment is desirable and in what fashion. Before exiting, disk buffers are written, and network connections subject to timeout are closed.

If *FAST* is NIL, LOGOUT first saves your virtual memory in a file. Change the file name by setting the UNIX environment variable LDEDESTSYSOUT. If this variable is not set, the file saved is ~/lisp.virtualmem (i.e., lisp.virtualmem on the user's home directory).

If *FAST* is T, Medley stops without writing the virtual memory file. It is not possible to resume execution in the same image.

(IL:SAVEVM) **[Function]**

Saves your state, but does not exit. It causes the current virtual memory image to be written to the location specified by the environment variable LDEDESTSYSOUT, if this variable is set; otherwise it is written to ~/lisp.virtualmem. This allows Lisp to continue. Execution in Medley continues after memory is saved; thus, SAVEVM operates as a sort of checkpoint of the current working state. SAVEVM can cause the following error:

File-System-Resources-Exceeded.

(IL:SYSOUT *FILE*) **[Function]**

Performs the equivalent of SAVEVM and then copies the saved image to *FILE* for devices other than {DSK} and {UNIX} (e.g., XNS file servers). (See Chapter 5, Medley File Systems, for further information on {DSK} and {UNIX}.) SYSOUT can cause the following error:

File-System-Resources-Exceeded.

Sun-Specific Environment Functions

System Environment Functions and Variables

These functions, which interrogate the system environment, operate as described below when they are invoked on the Sun Workstation:

(IL:REALMEMORYSIZE) **[Function]**

On some machines, returns the total amount of real memory available; does not work on a Sun Workstation (i.e., returns a meaningless value).

(CL:MACHINE-TYPE) **[Function]**

Returns a string identifying the type of computer hardware the system is running under. On the Sun–3 workstation MACHINE-TYPE returns "mc68020". On a Sun–4 workstation, MACHINE-TYPE returns the string "sparc".

(IL:MACHINETYPE) **[Function]**

Identifies the generic type of Lisp machine in use. On the Sun Workstation, it returns the symbol IL:MAIKO.

(CL:MACHINE-VERSION) **[Function]**

Returns a string identifying the version of the emulator running; e.g., "Microcode version: 279, memory size: 16384".

(CL:MACHINE-INSTANCE) **[Function]**

Returns a string containing the workstation host ID (in hexadecimal) and the host name.

IL:LISP-RELEASE-VERSION **[Variable]**

Identifies the release number within a single major release name. In Medley 2.0, IL:LISP-RELEASE-VERSION is 2.0 While IL:MAKESYSNAME does not change, IL:LISP-RELEASE-VERSION always changes with each new sysout release. This variable did not exist in the Medley 1.0-S sysout.

IL:\MY.NSADDRESS **[Variable]**

Fills in the fields of the network address with the host ID if Medley is run without the Ethernet enabled. Programs that use the network address as a unique identifier should be aware that the value could vary from session to session depending on whether or not the Ethernet is enabled. (Refer to Chapter 14 of the *Interlisp-D Reference Manual* for further information.)

VM Functions

The biggest difference is a change in terminology. On Xerox 1100 series workstations, Lisp itself handles all virtual memory operations directly, so the terms "sysout" and "virtual memory image" can be used interchangeably. The running sysout resides in a reserved area on the workstation local disk (the virtual memory partition) that Lisp reads from and writes to as it needs to move pages into and out of physical memory.

(IL:VMEMSIZE) **[Function]**

Returns the number of 512–byte pages of the Medley virtual memory that are in use. This number is a good estimate of the size of a SYSOUT, MAKESYS, or SAVEVM virtual memory file.

(IL:VMEM.PURE.STATE *ON/OFF***)** **[Variable]**

Has no effect on the Sun Workstation. The virtual memory file is not modified except by an explicit (LOGOUT) or (SAVEVM).

IL:BACKGROUNDPAGEFREQ **[Variable]**

Has no effect on the Sun Workstation. The virtual memory file is not modified except by an explicit (LOGOUT) or (SAVEVM).

You can control how much virtual memory Medley uses by using the -m switch, as described below.

ldeether [<*SYSOUT-name*>] [-m<*memory-size*>] [*other options*] **[Command]**

Allows you to specify an arbitrary virtual memorey size for Medley.

-m Specifies the memory size

memory-size 8 through 32 Mbytes

When you use -m, the value of IL:\STORAGEFULLSTATE in the sysout you start should not be 3 or 4. Those values mean it already used more than the 8-Mbyte space in the sysout. Because of the Medley storage management architecture, the virtual memory size cannot be changed after IL:\STORAGEFULLSTATE has been set to 3 or 4. This value can be examined just before (IL:LOGOUT) if you want to specify the virtual memory size during the next start-up.

Example: `ldeether /usr/LISP.SYSOUT -m 16`

This example means 16 Mbytes of virtual space will be assigned for Lisp.

Stopping Lisp Temporarily

(IL:SUSPEND-LISP) **[Function]**

Suspends, temporarily, the UNIX process running Medley. Using the `fg` C-Shell command, the Medley process can be continued from the C-Shell where it was started. `SUSPEND-LISP` has no effect on Xerox Lisp workstations. This function should not be used during I/O operations (file or network).

Login Functions

This section describes the interaction between the usernames and passwords in Medley and the SunOS usernames and passwords. The functions `IL:USERNAME`, `IL:SETUSERNAME`, `IL:SETPASSWORD`, and `IL:LOGIN` access the username/password database used by Medley in network operations. (For further information, see Chapter 24 of the *Interlisp-D Reference Manual*.) When Medley is started, this database contains only the SunOS username, with no password. Except for this, there is no interrelation between these Medley functions and SunOS usernames and passwords.

`IL:USERNAME` returns the SunOS login name under which the emulator was started. A subsequent `IL:SETUSERNAME` or `IL:LOGIN` changes `IL:USERNAME`, and the default login name for network access to XNS and PUP hosts. However, it does not change the SunOS login name or access capabilities for files on {`DSK`} or {`UNIX`}. (See Chapter 5, Medley File Systems, for detailed information on {`DSK`} and {`UNIX`}.) Because it doesn't change the SunOS login name, it won't change the author name on SunOS files created from Lisp.

The following functions apply to login activities.

(IL:UNIX-USERNAME) **[Function]**

Returns a string consisting of the username of the SunOS process running Medley. Returns `NIL` if one of the following conditions apply:

- You are not running under UNIX
- You do not have a full name entered in `/etc/passwd` or the NIS password map
- An error occurs.

(IL:UNIX-FULLNAME) **[Function]**

Returns a string containing the full name of the owner of the SunOS process running Medley. Returns NIL if the user is not running under UNIX or an error occurs.

(IL:LOGIN *HOST FLG DIRECTORY MSG***)** **[Function]**

Attempts to maintain user IDs and passwords for network as well as local access. If *HOST* is NIL, this function attempts to perform the SunOS `setuid` operation.

Unless you are running as root, this will not change your SunOS login.

Environment Inquiry

The following functions return the values of UNIX environment variables or machine parameters. They return NIL if run in Medley on Xerox 1100 series workstations.

(IL:UNIX-GETENV *STRING***)** **[Function]**

Returns the value of the environment variable with the given name. The argument *STRING* should be the name of a UNIX environment variable. For example, `(UNIX-GETENV "HOME")` might return the user's home directory.

(IL:UNIX-GETPARM *STRING***)** **[Function]**

Returns the value of one of a few built-in parameters. The argument *STRING* should be the name of one of the following UNIX environment variables:

Variable	If running on this hardware	Returns
"MACH"	Sun-4	"sparc"
	Sun-3	"mc68000"
	RS/6000	"rs/6000"
	HP9000	"hp9000"
	DEC3100	"mips"
	PS/2	"i386"
"ARCH"	Sun-4	"sun4"
	Sun-3	"sun3"
	RS/6000	"rs/6000"
	HP9000	"hp9000"
	DEC3100	"dec3100"
	PS/2	"ps/2"
"HOSTNAME"	All	Returns the local host name
"HOSTID"	All	Returns the local host identification number as a hexadecimal string

Display and Keyboard Functions and Variables

Some Medley display and keyboard functions and variables operate differently on the Sun Workstation.

The following functions have no effect on a Sun Workstation, and always return NIL:

IL:CHANGEBACKGROUNDBORDER

IL:VIDEORATE

IL:SETMAINTPANEL

IL:VIDEOCOLOR

The functions IL:BEEPON, IL:BEEPOFF, IL:PLAYTUNE, IL:RINGBELLS generate monotones.

(IL:BEEPON *FREQ***)** **[Function]**

Turns on the keyboard tone generator on the Sun Workstation. The *FREQ* argument is ignored.

(IL:BEEPOFF) **[Function]**

Turns off the keyboard tone generator.

(IL:PLAYTUNE *TUNEPAIRS***)** **[Function]**

Sounds tones, but ignores the frequencies of the values in *TUNEPPAIRS*.

(IL:RINGBELLS) **[Function]**

Causes the machine to beep several times.

Timers and Clocks

UNIX is a timesharing operating system. When Medley is running, other programs can be running at the same time on the same workstation.

On a Xerox workstation running Lisp, CPU time could be computed exactly from elapsed time after subtracting known system overhead. To allow older Interlisp-D programs to work unchanged, the timer functions were modified to allow programs that accounted for time on Xerox workstations to continue to run. Time is categorized as follows:

CPU time:	The total amount of time spent executing Medley's process in user mode.
SWAP time:	The total time spent running other processes (Elapsed time – (CPU time + Disk time).
Disk I/O time:	The total amount of time spent in the system executing on the behalf of Medley's process.

The Medley functions `CLOCK`, `TIME`, and the like get the time of day directly from SunOS. The function `SETTIME` has no effect on the Sun Workstation.

IL:\RCLKMILLISECOND **[Variable]**

The number of clock "ticks" in a millisecond. On the Sun Workstation, this value is always 1000. All of the timer functions that deal in clock ticks will do their computation in microseconds. Note, however, that the Sun Workstation does not have that accurate a clock resolution. While clock resolution varies from one operating system version to another, it often has a resolution no better than 1/60th of a second.

Miscellaneous Operational Differences

The stack and virtual memory handling functions on the Sun Workstation are implemented differently from the way they are on the Xerox workstations. For this reason, the "cursor bars" used on the Xerox workstations are not used on the Sun Workstation.

When working in Medley on a Sun workstation, you should periodically load a fresh sysout. Older Medley sysouts don't run as well as "fresh" sysouts due to a number of factors such as fragmentation of memory, increased working set, more objects taking up various spaces (e.g., gc tables), reduced symbol space.

On Xerox workstations, users are reminded to reload fresh sysouts, because they eventually fill up their sysout partition. With Sun workstations, there is no such limit reminder, so users' sysouts tend to grow to the maximum size (32 MB), and thus run slower and slower.

Console Messages

Under SunOS, various system processes and operations attempt to log information on the console. Since Medley takes over the screen, console messages are redirected (except when running under X); a background process in Medley causes them to appear in the prompt window.

However, when Medley is run remotely (i.e., not from the console), most console, or operating system, messages are printed in the prompt window. However, some messages may also appear in the middle of the Medley display screen or on the remote tty. This occurs because UNIX is often confused about where to send messages. Note that Medley is normally run remotely only for debugging purposes.

CAUTION

Critical UNIX system processes can hang if the buffer holding console messages fills. Medley uses a temporary file, `/tmp/XXXX-lisp.log`, where `XXXX` is the user's login name, to buffer console messages before printing them. Do not delete this log file while Medley is running. If the log file is deleted, console messages can no longer be printed in the Medley prompt window.

[This page intentionally left blank]

5. MEDLEY FILE SYSTEMS

This chapter discusses the conventions for using files from Medley.

File Naming Conventions

In Lisp, a file name (pathname) consists of a collection of fields: the *host, directory, name, extension* and *version*. These fields are optional. The standard Lisp syntax for these fields is:

{host}<directory>name.extension;version

The *directory* field can be a directory path consisting of a sequence of directory and subdirectory components. Slashes (/) and right angle brackets (>) can be used to delimit a directory name; there is no distinction made between them. Square brackets ([]) are not acceptable as directory delimiters.

Duplicated directory delimiters are treated as a single delimiter. Thus, the following two file names specify the same file:

```
{DSK}<LISP>USERS>FOO.;1
{DSK}</LISP/USERS/>FOO.;1
```

Hosts that Medley Supports

{CORE}	Creates "files" in memory; useful for quick temporary files
{LPT}	Creates files that are automatically sent to your printer
{NULL}	Creates a file that does nothing
{DSK} and {UNIX}	Give you access to the Sun's file systems; the rest of the chapter concentrates on them.

The above hosts are described in more detail in the IRM.

Using SunOS Files from Medley

You can access any mounted SunOS file system directly from Lisp. The mounted file system is available as an I/O device of the Lisp environment. This file system appears as the local disk of Lisp, even though it may be a remotely mounted file system of networked Sun file servers.

Many of the file devices to which the Medley environment can talk, including PUP, XNS file servers, the {CORE} device, and others, have facilities that are not directly supported by SunOS. For example, many file systems have file version numbers and case insensitive file search conventions.

Medley on the Sun Workstation has two distinct "host" names that can be used to access the SunOS file system. These host names are provided for compatibility with existing applications and tools. They also simultaneously allow natural interaction with the SunOS file system. The names are:

{DSK} On the Xerox workstation, {DSK} gave you access to your local hard disk; to use {DSK}, you had to create a directory on each disk partition you wanted to use. On the Sun Workstation, in contrast, the {DSK} device lets you access the file system using similar conventions to those used for {DSK} on the Xerox workstation local disk devices. In particular, {DSK} files have version numbers; {DSK} file name recognition also ignores the case of letters.

{UNIX} The {UNIX} device lets you use the mounted file systems with the normal naming conventions of the SunOS file system. {UNIX} files do not have version numbers, and the file name recognition treats lowercase letters as distinct from their uppercase equivalents.

File streams can be opened or closed on both devices. The reason for having both devices is to more easily support the running of applications that were originally developed on a Xerox workstation, while still allowing new applications to interact more naturally with UNIX.

NOTE: Both {DSK} and {UNIX} work as filters. They act as pointers to a device. On 11xxs, {DSK}foo is the same as {DSK}<lispfiles>foo. On the Sun, {DSK}foo is the same as $HOME/foo ~*user*/foo.

Common {DSK} and {UNIX} Naming Conventions

- To include a special character (e.g., > or ;) in a file name, precede it with a single quote (’). To include a single quote in a file name, precede it with another single quote. You can quote any of these characters: <, >, ;, ~, and a period (.). The following examples show how the single quote notation on {DSK} and {UNIX} is used.

{DSK} Name From Lisp	**File Name From SunOS**
foo’>bar.baz;1	foo>bar.baz
foo’;bar.baz;1	foo;bar.baz
foo’’bar.baz;1	foo’bar.baz

- {DSK} and {UNIX} do not allow you to use either the slash (/) or the NUL character in file names. Thus, you cannot name files containing these characters.

- Both {DSK} and {UNIX} can handle the following characters, which were defined as special characters in Medley Release 1.1: backslash (\) and tilde (~).

- {DSK} and {UNIX} can distinguish between a file name with a period at the end (e.g., foo.) and a simple file name (e.g., foo). The final period is preceded with a single quote, as shown in the following example:

{DSK} Name From Lisp	**File Name From SunOS**
foo.;1	foo
foo’..;1	foo

- On {DSK} and {UNIX}, the C-Shell and SunOS directory notations (~, ., and ..) are supported in the Lisp directory specification. The tilde character (~) is allowed at the very beginning of the directory specification of a pathname. A combination of relative path specifiers (~, ., ..) is supported. The tilde character corresponds to the user's home directory at login. The period (`.`) corresponds to the current working directory. Two periods (`..`) indicates the parent of the current working directory.

- File names are returned by the system (e.g., `INFILEP`) in more canonical form. The function which returns the full file name returns it in the canonical form , as in `{DSK}<usr>etc>` rather than `{DSK}/usr/etc/`. This change will make some tools which depend on the conventional file name representation described in the *Interlisp-D Reference Manual* work correctly on the Medley file system (e.g., COPYFILES).

{DSK} Naming Conventions

The `{DSK}` device performs the following file name transformation when actually accessing the SunOS file system:

- Mixed case letters are read as such.
- File name searches are done case-sensitive first; if a match is not found, the system does a case-insensitive search.
- The left angle bracket character (<) is translated to a slash (/), the delimiter for the root directory.
- `{DSK}` supports relative pathnames. You can specify relative pathnames by omitting a slash (/) or left angle bracket (<) as the first character in the directory field. For example:

  `{DSK}foo.fee` and `{DSK}~/foo.fee` are relative to the user's UNIX home directory (`~/foo.fee`).

  `{DSK}./foo.fee` is relative to the user's current working directory (`SunOS./foo.fee`).

  `{DSK}../foo.fee` is relative to the parent directory of the user's current UNIX working directory (`../foo.fee`).

The Medley 2.0-S `{DSK}` device supports the notation in which the three meta characters (`'`, `..`, and `~`) are used together, as shown in the following example:

    {DSK}~/../tom/foo.c

In this example, `the` `{DSK}` device interprets `tom` as one of the subdirectories of the parent directory of the user's home directory.

`{DSK}` also supports the tilde-name (~name) convention. `{DSK}` interprets `{DSK}~tom/foo.c` as a file named `foo` on `tom`'s home directory. In this notation, the user name is case-sensitive (e.g., `~tom` and `~Tom` are treated as different users).

Version Numbering

The UNIX file system does not support version numbers in file names; `{DSK}` emulates versions with a naming convention. (GNU Emacs also uses this convention.) This section explains how `{DSK}` version numbers are represented in the SunOS file system.

- When you create a completely new file, it appears in the SunOS file system without a version number.

{DSK} Name From Lisp	**File Name From SunOS**

`bar.baz;1`	`bar.baz`

- When you create (from Medley) a file with a version other than 1, Medley adds version numbers to that file name, as a trailing number between tildes, e.g., "`myfile.~12~`" for the twelfth version of myfile.

  The following shows some examples of equivalent file names in Lisp and SunOS.

{DSK} Name From Lisp	**File Name From SunOS**
`bar.baz;1`	`bar.baz.~1~`
`bar.baz;2`	`bar.baz.~2~`
`bar.;23`	`bar.~23~`

- Medley always maintains a versionless file which is hard-linked to the highest extant version of the file (i.e., they are two names for the very same file). This file name does not appear in the {DSK} directory listing.

From {DSK}	**From SunOS**
`foo.c;15`	`foo.c` (hard linked with `foo.c.~23~`)
`foo.c;23`	`foo.c.~15~`
	`foo.c.~23~`

  Similarly, a file created in UNIX with no version number is treated by {DSK} as the highest version.

- When you create a new version of a file, the versionless–file link is broken, and the versionless file is hard–linked to the new highest version.

From {DSK}	**From SunOS**
`foo.c;15`	`foo.c` (hard linked with `foo.c.~24~`)
`foo.c;22`	`foo.c.~15~`
`foo.c;24` (new file)	`foo.c.~22~` (no link with `foo.c`)
	`foo.c.~24~` (new file, link from `foo.c`)

- When you delete the highest version of a file, the versionless file is also deleted. If any older versions of the file remain, a new link is created from the versionless name to the highest version extant. For example, if you have the files

From {DSK}	**From SunOS**
`foo.c;1`	`foo.c` (linked to `foo.c.~2~`)
`foo.c;2`	`foo.c.~1~`
	`foo.c.~2~`

  and you delete `foo.c;2` from {DSK}, the resulting files are:

From {DSK}	**From SunOS**
`foo.c;1`	`foo.c` (linked to `foo.c.~1~`)
	`foo.c.~1~`

- When you rename a file, it works the same as deleting the file under the old name then creating it under the new name. For example, if you have the following {DSK} files

From {DSK}	**From SunOS**
`foo.c;1`	`foo.c` (linked to `foo.c.~2~`)
`foo.c;2`	`foo.c.~1~`
`fee.c;1`	`foo.c.~2~`
`fee.c;2`	`fee.c` (linked to `fee.c.~2~`)

```
fee.c.~1~
fee.c.~2~
```

and you rename "foo.c" to "fee.c", your renamed {DSK} files and the linked SunOS files would appear as:

From {DSK}	**From SunOS**
foo.c;1	foo.c (linked to foo.c.~1~)
fee.c;1	foo.c.~1~
fee.c;2	fee.c (linked to fee.c.~3~)
fee.c;3	fee.c.~1~
	fee.c.~2~
	fee.c.~3~ (renamed file)

- When a file has a name suffix that is not a valid version number (e.g., myfile.~12x~), the suffix is regarded as part of the file name.

From {DSK}	**From SunOS**
myfile.~12x~;1	myfile.~12x~

Pathnames

A pathname on {DSK} is always case insensitive. When the user specifies a file, the {DSK} device handler first searches for the file with the specified name. If no such file is found, it then searches for a file with the same spelling but different case.

Most Lisp functions, such as FINDFILE and INFILEP,which return pathnames return them with the original case when they are applied on files on {DSK} when IL:*DSK-UPPER-CASE-FILE-NAMES* is NIL. If IL:*UPPER-CASE-FILE-NAMES* is not NIL, these functions return only uppercase pathnames. The only exception is the function DIRECTORY , which returns a list of pathnames. The case of the pathnames is controlled by the global variable IL:*UPPER-CASE-FILE-NAMES* in a similar manner to IL:*DSK-UPPER-CASE-FILE-NAMES*.

If a pathname on {DSK} has no directory specification, a tilde-slash combination (~/) is used, i.e., the Lisp directory specification {DSK}foo is the equivalent of {UNIX}~/foo.

{UNIX} Naming Conventions

For the {UNIX} device, file name translation takes place only on the directory. An initial left angle bracket (<) is treated as if it were an initial slash (/); both signify a path relative to the SunOS file system root directory; if there is no initial left angle bracket or slash, the directory is relative to the current working directory. Initially this is the working directory where Lisp was started; you can change it using the CHDIR function, described below. Tilde (~) is translated to the user's home directory.

For example, {UNIX}myfile/abc means the file abc on the ./myfile directory.

The {UNIX} device does not recognize version numbers, does not return them, and ignores them for recognition.

No case translation or recognition is done; upper- and lowercase letters are treated as distinct.

Examples:

{UNIX} Name From Lisp	**File Name From SunOS**
`<foo>fee>bar.baz;1`	`/foo/fee/bar.baz;1`
`<foo>fee/bar.;1`	`/foo/fee/bar.;1`
`<foo/fee>`	`/foo/fee/`
`</foo/fee/>`	`/foo/fee/`
`/foo/fee/bar.~1~`	`/foo/fee/bar.~1~`
`/foo/fee/`	`/foo/fee/`

In the first two examples the `;1` is treated as part of the file name, not the version number. In the last two examples that translation is not done.

Directories

In places where Lisp expects a directory name, `{UNIX}` paths must end with a slash (/).

Directory Enumeration

You cannot use the wildcard character, asterisk (*), in subdirectories for either `{DSK}` or `{UNIX}` devices. For example

```
>(DIRECTORY '{DSK}/users/x*/foo)
NIL
```

Enumeration of files in directories differs between `{DSK}` and `{UNIX}` devices. On the `{DSK}` device, a versionless file which has a link to the highest version file is not enumerated in a directory.

On the `{UNIX}` device, all files are enumerated in a directory. For instance, if the following SunOS files linked with `foo.c.~2~` exist

```
foo.c
foo.c.~1~
foo.c.~2~
```

the `{DSK}` directory enumeration would look like this:

```
>(DIRECTORY '{DSK}/users/venue/*)
({DSK}/users/venue/foo.c;1
{DSK}/users/venue/foo.c;2)
```

The `{UNIX}` directory enumeration, on the other hand, would look like this:

```
>(DIRECTORY '{UNIX}/users/venue/*)
({UNIX}/users/venue/foo.c
{UNIX}/users/venue/foo.c.~1~
{UNIX}/users/venue/foo.c.~2~)
```

Directory Creation

`{DSK}` When you write a new file on `{DSK}`, if the directory named in a pathname does not exist, the `{DSK}` device handler creates the directory automatically. This feature is provided for compatibility with other Interlisp-D implementations.

If you try to "connect" to a nonexistent directory (using the CONN Exec command or the function IL:CNDIR), Medley returns the message

```
Nonexistent directory
```

{UNIX} The {UNIX} device does not support such directory creation. An attempt to create a file on a nonexistent directory results in an error.

```
{UNIX}/users/venue/foo.c.~2~)
```

Directory Deletion

Neither {UNIX} nor {DSK} support automatic directory deletion. To delete a directory you must use the SunOS C-Shell command rmdir.

Open File Limit

The number of simultaneously open {DSK} and {UNIX} files must fall within the SunOS limits for a process. For OS 3.4, this number of open files may be configured, with 30 as the maximum permissible number of open files per process. This means that it is not possible to have more than 30 files open for a process, minus whatever files Medley has open for its own use, at any one time in the Medley system. If you try to open too many files, the system call error number 24, Too many open files, appears in the prompt window.

For OS 4.0, the maximum number of files/processes that can be open at one time is 64, unless your kernel is configured otherwise.

Default Pathname

If no path is given, the {DSK} device defaults to the user's home directory, tilde-slash (~/). The {UNIX} device defaults to the current working directory. This current working directory can be changed with the CHDIR function. The current working directory is also used to resolve the interpretation of the period (.) and double period (..) specifications at the beginning of a {DSK} pathname.

(IL:CHDIR ***PATHNAME*****)** **[Function]**

Changes the current working directory for the current invocation of Lisp. For example,

```
(CHDIR "{DSK}~/subdir/")
(OPENSTREAM "{DSK}./foo" ...)
```

opens the SunOS file ~/subdir/foo.

When *PATHNAME* does not end with a slash (/) or right angle bracket (>), the whole *PATHNAME* is treated as a directory name:

```
(CHDIR '{DSK}<users>local>)
> "{DSK}<users>local>"
(CHDIR '{UNIX} /usr/local)
> "{UNIX}<usr>local>"
```

If *PATHNAME* is NIL, CHDIR tries to change the current working directory to the current connected directory. If the directory is connected to devices other than {DSK} or {UNIX}, the error message

```
Bad Host Name
```

appears, followed by the host name of the current connected directory.

If the *PATHNAME* does not exist, the error message

`No-Such-Directory`

appears followed by the system echo of the pathname.

File Attributes

This section describes how the various file attributes are treated by Lisp on the Sun Workstation and what they translate to in SunOS.

GETFILEINFO obtains file attributes and SETFILEINFO sets the attributes.

WRITEDATE and CREATIONDATE **[File Attributes]**

Resets the date to the current time whenever the contents of a file are modified. This only works for the owner of the file. Since UNIX does not naturally support more than one date for file modification, the WRITEDATE and CREATIONDATE are treated identically by Lisp functions OPENSTREAM, OPENFILE, GETFILEINFO, and by the {DSK} and {UNIX} devices.

TYPE **[File Attribute]**

Sets the TYPE property of files; normally either TEXT or BINARY. However, UNIX does not distinguish between TEXT and BINARY files. Normally, programs will infer the type by the file extension, using the Lisp variables DEFAULTFILETYPE and DEFAULTFILETYPELIST. This is the convention used by Medley. If no file extension is given, the value in DEFAULTFILETYPE is used. SETFILEINFO cannot change the TYPE attribute.

EOL **[File Attribute]**

Returns the end–of–line convention. Both the {DSK} device and {UNIX} use line feed (LF) as the default EOL convention for text. The EOL for binary files is carriage return (CR). EOL uses the TYPE property of files. (The TYPE property of a file depends on the file extension and the DEFAULTFILETYPE and DEFAULTFILETYPELIST variables). If the TYPE property of a file is TEXT, LF (=10) is used as EOL. If the TYPE property of a file is BINARY, CR (=13) is used as EOL.

NOTE: EOL conventions on {DSK} are not compatible with those on Xerox workstations.

AUTHOR **[File Attribute]**

Returns the author of the file, i.e., the login name of the user who created it. This attribute cannot be changed.

PROTECTION **[File Attribute]**

Returns file protection attributes. The file protection attributes of files under the SunOS cannot be directly manipulated from inside Lisp. It is necessary to use the UNIX chmod command to change file protection bits.

SIZE **[File Attribute]**

Returns the file size. For compatibility with other Lisp environments running on Xerox workstations, the SIZE attribute is computed as the length of the file (in bytes) divided by 512 (rounded up).

NOTE: SETFILEINFO lets you change the SIZE attribute of I/O streams and output streams. However, a file cannot be expanded this way.

File Variables

This section discusses how certain file variables are used by Medley in SunOS.

IL:FileTypeConfirmFlg **[Variable]**

The file–type attribute of a file on {DSK} or {UNIX} is decided from its extension, DEFAULTFILETYPELIST and DEFAULTFILETYPE. Extensions of binary files should be registered in DEFAULTFILETYPELIST. When this rule is broken, a hardcopy of files on {DSK} and {UNIX} may confuse the printers. So when you try to hardcopy a file whose extension is not registered in DEFAULTFILETYPELIST, a menu is invoked to confirm the file type. Text or binary can be selected. The invocation of this menu can be stopped by setting IL:FileTypeConfirmFlg to NIL. The default value of IL:FileTypeConfirmFlg is T.

When extensions of binary files are not registered in DEFAULTFILETYPELIST, copy or rename from a DSK/UNIX device to a non–DSK/UNIX device also may cause file type confusion. This type of copy or rename results in one of the following warning messages in the prompt window, as appropriate:

Extension of {DSK}foo.fee;1 isn't in DEFAULTFILETYPELIST. {CORE}foo.fee;1 was copied as TEXT.

This message can be stopped by set FileTypeConfirmFlg to NIL.

Extension of {DSK}foo.fee;1 isn't in DEFAULTFILETYPELIST. {CORE}foo.fee;1 was renamed as TEXT.

This message can be stopped by set FileTypeConfirmFlg to NIL.

Either of these messages can be stopped by setting IL:FileTypeConfirmFlg to NIL.

IL:DEFAULTFILETYPE **[Variable]**

Initially set to TEXT. Used with the file attribute TYPE.

DEFAULTFILETYPELIST **[Variable]**

A list of accepted file types. Initially set to ((NIL . TEXT) (C . TEXT) (H . TEXT) (LISP . TEXT) (LSP . TEXT) (O . BINARY) (OUT . BINARY) (LCOM . BINARY) (DFASL . BINARY) (DCOM . BINARY) (SKETCH . BINARY) (TEDIT . BINARY) (DISPLAYFONT . BINARY) (WD . BINARY) (IP . BINARY) (RST . BINARY) (BIN . BINARY) (MAIL . BINARY) (SYSOUT . BINARY))

Used with the file attribute TYPE. Binary files, such as Sketch files, InterPress files, or Press files, should have their extensions registered in DEFAULTFILETYPELIST. This is especially important because UNIX does not support file types.

File System Errors

Several types of errors may occur in the Medley file system.

When a remotely mounted file system or NFS service is down, or when network traffic is heavy, any attempt to access a file on that file system results in an error. The following error message is printed in the prompt window:

```
File access timed out
```

Medley will wait until the file system responds or until a timeout occurs. If the file system is mounted with the "hard" option, the timeout is controlled by the value of environment variable `LDEFILETIMEOUT`. If the file system is mounted with the "soft" option, the timeout depends on the NFS file system timeout time, and the value of `LDEFILETIMEOUT`. Medley will wait until the shorter of these two times is exceeded. The NFS file system timeout time, retry times, etc., are controlled by the UNIX command **mount**.

If `LDEFILETIMEOUT` is not set, the default value of 10 seconds is used. The variable is inspected at boot time, and a setting between 1 and 100 seconds is appropriate in most cases.

The following error messages may appear when there are Medley file errors:

```
Not owner
Device error:
Protection-violation
File-won't-open
Too-Many-Files-Open
File too large
File-System-Resources-Exceeded
Connectiom timed out
No-Such-Directory
Bad Host Name
FS-RENAMEFILE-SOURCE-COULDNT-DELETE
```

Another type of error occurs when the user has insufficient access to files. When this happens, Medley will print the following message:

```
File not found
```

The following message then appears in the SunOS prompt window:

```
System call error:  open errno=13  Permission denied
```

See the *UNIX Interface Reference Manual*, Intro (2), for descriptions of all OS system call messages.

[This page intentionally left blank]

6. ERROR RECOVERY

Medley on the Sun Workstation has an error handling system which includes the following:

- The Xerox Lisp error system, described in the IRM
- A diagnostic program, URAID, which handles emulator errors

Occasionally, you may encounter SunOS error messages. Refer to your Sun documentation set for recovery procedures when these errors occur. When running Medley on a Sun Workstation, previous Lisp error handling such as Teleraid and MP errors are no longer available. However, you can still use Teleraid from a Sun Workstation to debug a Xerox 1100 series workstation.

URAID

The Medley system normally operates as a self–contained environment. In some unusual circumstances Medley may encounter a situation from which it cannot recover. In this case, when an unrecoverable emulator error is encountered, the emulator halts and enters into a small debugger called URAID. URAID allows you to inspect memory, or to look inside the sysout file, and attempt to recover from the error.

If you produce the same type of error condition in Medley on a Sun Workstation as you did on a Xerox workstation, you get a URAID error instead of an MP error.

Entering URAID

Normally, the emulator automatically enters URAID when an unrecoverable emulator error occurs. However, there are two additional methods available when you want to enter URAID directly.

- Use the SHIFT-CTRL-DELETE key combinations to enter URAID between opcodes. Note that the DELETE key referred to here is in the L10 position on the left keypad of the Sun keyboards. This sequence allows you to return to Lisp later.
- Use SHIFT-CTRL-NEXT for emergency interrupts only. Note that the NEXT key is in the ALTERNATE key position on the regular Sun keyboard. These combinations are useful for exiting from an opcode infinite loop. SHIFT-CTRL-NEXT does not necessarily enter URAID between opcodes; once you are in URAID mode, another URAID command such as "**f**" could cause the emulator to crash. At this point it is unlikely that you could return to Lisp. USE WITH CAUTION!

Conventions

URAID uses these display conventions:

- Numbers are displayed in hexadecimal unless otherwise noted.
- The *litatom* should be an uppercase string when used with a package prefix (e.g., `XCL:EVAL`).
- Symbols are displayed with a package prefix, but with no escape character.

- Symbols in the Interlisp (`IL:`) package are case-sensitive (e.g., `IL:\InterfacePage`); symbols in other packages are case-insensitive (e.g., `XCL:EVAL`).

In addition, these input conventions apply:

- Symbols may only be qualified by their home package.
- A full package name may prefix an input symbol. URAID also supports approved abbreviations of package names (e.g., `XCL:`, `SI:`, `CL:`, `XCLC:`).

  A symbol without a prefix is treated as a symbol in the Interlisp package. For instance, `\InterfacePage` is the equivalent of `IL:\InterfacePage`.
- Type-in is uppercase for symbols in any package except the Interlisp package; type-in is in mixed case for `IL:` package symbols or symbols with no prefix.

URAID Commands

URAID has a few simple commands which you can use to attempt diagnosis and error recovery. All URAID commands are case-sensitive.

h Hard Reset. Attempts to recover by resetting the Lisp stack. Quits URAID and causes Lisp to resume execution. This command should not be used unless you are sure that execution can be resumed.

e Exits to SunOS. Medley will end.

q Quits URAID and returns to Lisp.

NOTE: An error may occur while the Medley system is running uninterruptably. The following message signals this error:

```
Error in uninterruptable system code -- ^N to continue into
error handler
```

Disregard the `^N` command; it is not supported by URAID. Use the **q** command to continue.

Displaying a Stack

For casual users, the **l** command followed by several **f** commands generally provides the most useful information. Many of the other commands require some knowledge of the internal representation of Lisp objects and stack frames.

c Checks all user stack contents; stack inconsistency is displayed.

k *type* Changes the stack link that precedes the **l** command to be *type*, which is either **a** (to follow ALinks) or **c** (to follow CLinks). The default is to trace ALinks. ALinks follow the chain of free variable access.

l *type* Shows the stack as a back trace consisting of a numbered sequence of frame names. The default is the user stack. The argument *type* is a single letter denoting the stack to view. The system has a number of special contexts, which are areas of stack space used by certain system routines. Legal values of *type* are as follows:

g (garbage collect)
k (keyboard handler)
m (miscellaneous)
p (page fault)
r (reset)
u (user stack) - Default

type := g | k | m | p | r | u or nil

C Checks the contents by scanning all stack space in the sysout. For example:

0x11880 BF,[ivar:0x1800]
0x11802: FX for `CL:T`[ ]
0x11816 BF,[ivar:0x1816
0x11818: FX for `IL:\TURN.ON.PROCESSES`[ ]

Viewing Frames From a Stack

After displaying a particular stack with the **l** command, the following commands view individual frames from that stack:

f *number* Displays the contents of frame *number* (decimal) with its basic frame, IVars and PVars. The frame is printed in two parts, a basic frame containing the function's arguments and a frame extension containing control information, the function's local (PROG) variables, and dynamic values. On the left side of the printout are the hexadecimal contents of each cell of the frame, with an interpretation, usually as a Lisp value, on the right. The following message appears as you display a frame with the **f** command:

```
Press Return (To quit ESC and RET)
```

To abort the printing of a frame, first press the ESC key then the RETURN key. The URAID prompt "<" reappears.

<CR> Displays the next frame (closer to the root, or bottom, of the stack). This is the same as **f** *n*+1, where *n* is the number of the frame most recently viewed. Immediately after an **l** command, *n* is zero, so **<CR>** views the first frame.

a *litatom* Displays the top-level value of the *litatom*

d *litatom* Displays the contents of definition cell for the *litatom*. If it is compiled code, this command prints a CCODEP hexa-decimal address pointer; for example,

```
{CCODEP}0x14ccc4
```

Otherwise, it prints a Lisp definition; for instance, interpreted code returns

```
( LAMBDA () ...)
```

M Displays TOS, CSP, PVar, IVar, PC.

m *func1 func2* Moves the definition of *func1* to *func2*.

t *Xaddress* Displays the type of this object.

p *litatom* Displays the contents of the *litatom*'s property list.

w Displays the current function name and PC.

x *Xaddress[Xnum]* Prints *Xnum* word (16-bits) of the raw contents of the virtual memory starting at virtual address *Xaddress*. This is most useful for examining the contents of a datatype which other commands simply print as its virtual address.

@*litatom[snumber | NIL | T]* Sets the TOPVAL of *litatom* to the specified value. *snumber* is a signed smallp number.

<*Xaddress val* Sets the the contents of the word (16-bits) at the *Xaddress* to *val*.

Miscellaneous

v *filename* Saves the current virtual memory on the *filename*. This file can be examined using the functions `READSYS` and `VRAID` in the TeleRaid Lisp Library module, but cannot be used as a sysout file.

NOTE: This sysout cannot be restarted.

s Invokes a subshell.

(*num* Sets the print level (default is 2).

? Displays this summary.

! Prints the error message passed from the emulator.

Other Fatal Error Conditions

Occasionally, other emulator, operating system, or system administration errors may occur from which the URAID program cannot recover. Such error conditions include the process dying, the emulator going into an infinite loop, the keyboard being lost, or the system freezing up.

If any of these emulator errors occur, use the UNIX `kill` command to kill the `lde` process.

Lisp Errors

Errors While Running Medley

The following Lisp errors may occur when running Medley on the Sun Workstation.

ERROR MESSAGE	**CAUSE**
`File access timed out`	Occurs when you try to access a file when the remotely mounted file system or NFS service is down, or when network traffic is heavy. See the File System Errors subsection of Chapter 5.
`File too large`	Self–explanatory.
`Too-Many-Files-Open`	Occurs when you exceed one of the following: • SunOS open file limit (see Chapter 5, Medley File Systems) • System file resources while writing a sysout (using `IL:SYSOUT`)

Nonexistent directory	Occurs when you try to connect to a nonexistent directory using the IL:CNDIR function or the CONN command.
No-Such-Directory	CHDIR
Connection timed out	Self–explanatory.
Bad Host Name	Self–explanatory.
FS-RENAMEFILE-SOURCE-COULDNT-DELETE	Occurs when you try to rename a file which exists on a directory or which you do not have delete permission.

Xerox Workstation-Specific Errors

These Xerox workstation–specific errors may occur if certain functions are inadvertently used on the Sun Workstation.

ERROR MESSAGE	CAUSE
Floppy: No floppy drive on this machine.	Self–explanatory.
Device error: {FLOPPY}	Occurs when the user tries to enter a Lisp floppy function while running on the Sun Workstation.
Wrong machinetype	Occurs when functions controlling Xerox disk drive device–specific behavior are entered while running in SunOS.

Virtual Memory Errors

ERROR MESSAGE	LISP FUNCTION RESPONSIBLE
File-System-Resources-Exceeded	IL:SYSOUT, IL:LOGOUT, IL:SAVEVM
Protection-Violation	IL:SYSOUT, IL:LOGOUT, IL:SAVEVM
File-Wont-Open	IL:SYSOUT, IL:LOGOUT, IL:SAVEVM

[This page intentionally left blank]

APPENDIX A. INSTALLATION HINTS

Medley Shell Variables

The following is a fragment of a `.cshrc` file which you may want to adapt to your own needs. In this example Smythe works in Building 12b (bldg12b), and always wants a fresh sysout, containing Rooms, loaded.

```
# ============================================
# Set up various Medley variables.
setenv LDEDESTSYSOUT /user/smythe/sysouts/saved.virtualmem
setenv LDESRCESYSOUT /usr/share/lde/lispsysouts/ROOMS.SYSOUT
setenv LDEINIT       /usr/share/lde/site-files/bldg12b-init.lcom
# Assuming you are using UNIXChat and VTChat,
    configure the Chat window
  if ($?LDESHELL == 1) then
      setenv TERM vt100
      stty erase ^H
  endif
# ============================================
```

Running on Multiple Workstations

Installation for Sites with Sun–3 and Sun–4 Workstations

In Medley 2.0, the only differences between the Sun–3 and Sun–4 distributions are in the `install.sunosX` directories. Thus, during installation the common subdirectories (`lispsysouts`, `lisplibrary`, `fonts`, etc.) might be installed instead to a shared file system, saving 15 MB of unnecessary duplicated space. In the example below, `/sharedserver` is a remote file system mounted on the local machine.

```
prompt% mkdir /sharedserver/lde
prompt% cd /sharedserver/lde
prompt% tar xvfb /dev/rXX0 126 ./lispsysouts ./lisplibrary
          ./fonts
```

If soft links are then left on `/usr/share/lde`, the installation can proceed as before.

```
prompt% ln -s /sharedserver/lde/lispsysouts
    /usr/share/lde/lispsysouts
prompt% ln -s /sharedserver/lde/lisplibrary
    /usr/share/lde/lisplibrary
prompt% ln -s /sharedserver/lde/fonts /usr/share/lde/fonts
```

Otherwise, the site initialization file needs to be changed appropriately.

The install directories are left on `/usr/share/lde`, since those directories are typically local to a particular processor architecture.

```
prompt% cd /usr/share/lde
prompt% tar xvfb /dev/rXX0 126 ./install.sunos4
```

Using a "runlde" on Multiple Workstations

The following is an example of a `runlde` script that might be used for running Medley on different machines.

```
# (invokes CSH)
# ============================================
# Usage: runlde optional-sysout
#
# The script below is for the following machines:
#
#      Host  HostID
#      ----  ------
#      timber      1700319b
#      gopher      17003016
#      tree  13003565
# ============================================

switch ("‘hostid‘")
      case ’1700319b’:
            ldeether $1 -k ’99e8bfc6 92299f45 9199a409’
            breaksw
      case ’17003016’:
            ldeether $1 -k ’70c5a8d8 7b0498cc 45e35500’
            breaksw
      case ’13003565’:
            ldeether $1 -k ’ce7627bf b5b61ac8 2f990cc0’
            breaksw
      default:
            echo  "Sorry,  host  ’‘hostname‘’  is  not  in  this  shell
      script"
      endsw
```

Configuring the Software

The software comes in these two forms:

- An executable binary image for users who have not modified the Sun kernel too extensively
- An object file that can be relinked for your particular system.

If you want to use the executable that Venue supplies, skip to the Enabling PUP/XNS Ethernet subsection below.

Relinking

If you have tried the prelinked software and it doesn't work, link the object code with the Sun libraries. To do this, you need the `suntool`, `sunwindow`, and `pixrect` libraries, and `make`, `cc`, etc., available on your search path. To configure the system, connect (`cd`) to the directory `usr/share/lde/install.sunosx` (where `x` is the version of SunOS that you are running, e.g., SunOS 4.0 in the following), and type make.

```
prompt% cd /usr/share/lde/install.sunos4
prompt% rm lde ldeether; make
```

This procedure replaces the two executable programs, `lde` and `ldeether`. The program `ldeether` enables access to Xerox network protocols from Lisp.

Enabling PUP/XNS Ethernet

If you intend to use the PUP or XNS Ethernet directly from Medley, you need to change file ownership and permissions of `ldeether`. Note that you do this on the server where `ldeether` is actually residing (root permission must be on the server). Log in to the machine where `ldeether` resides. To find out where `ldeether` resides, type:

```
prompt% df filename
```

where *filename* is the pathname of `ldeether`. The system responds with the name of a file system (e.g., `/dev/sd0g`) for a local file, or with a machine name and directory (e.g., `python:/user1`) for an NFS file.

Now you can change the `ldeether` file ownership and permissions.

```
prompt% rlogin server
server% su
server# cd /usr/share/lde/install.sunos4
server# /etc/chown root ldeether
server# chmod 4755 ldeether
server# exit
```

If you are using the Ethernet, substitute `ldeether` whenever `lde` appears in the instructions below.

Using NIS to Manage the Keys for Multiple Workstations

Here is an example how to handle several Medley licenses on a network, by using the Sun Network Information Service (NIS).

Create a file containing an association list of hostnames vs. license keys, for each host that has a Medley license. For example:

```
# medley-keys.by-hostname
# =========================================
king            6a1c33bf 11dc1a48 a4c34080
sidewinder      7b636e98 55a26cd4 26b80560
hognose         190750c0 17c658e0 08060ac0
boa             8334d182 00793e07 4903890b
asp             c90faa4f d3477c53 d304b85b
rattler         70b8fd18 2d79f344 c30051c0
```

NOTE that the following commands should all be run as `root`.

On your NIS master server, create an NIS database of hostname vs. Medley keys:

```
prompt% /usr/etc/yp/makedbm ./medley-keys.by-hostname \
                /var/yp/your-domain/medley-keys
```

Replace *your-domain* with the name of your NIS domain. The output is put in the directory containing your master NIS maps.

If you have NIS slave servers serving your domain, you will need to update each one manually the first time the map is created. Thereafter, they will be updated automatically. On each NIS slave server do the following:

```
% /usr/etc/yp/ypxfr -f -h your-NIS-master medley-keys
```

Replace *your-NIS-master* with the name your NIS master server.

After updating all NIS slave servers, you now need to propagate the NIS map to your NIS clients. On your NIS master, type:

```
% /usr/etc/yp/yppush medley-keys
```

From now on, any changes made to the `medley-keys.by-hostname` file will only require the propagation of the map to your NIS clients. The following steps are required:

1. Create a new NIS map using the `makedbm` command as described above.
2. Propagate the changes to your NIS clients using the `yppush` command as described above.

You can now use the newly created map. Below is an example of a `runlde` script that uses the newly created NIS map.

```
#! /bin/csh -f
# =========================================
# Usage : runlde [sysout]
#
# Script for running Venue Medley software.
#
# =========================================
if ($#argv > 1) then
    echo "Usage : runlde [sysout]"
endif

set SYSOUT = "$1"
set HOSTNAME = `/bin/hostname`
set KEY = `/bin/ypmatch $HOSTNAME medley-keys`
```

```
if ! $status then
    ldeether $SYSOUT -k "$KEY"
endif
```

Consult the *Sun Network and Communications Administration* manual for more details about NIS and how to add the new map to the `/var/yp/Makefile`.

[This page intentionally left blank]

APPENDIX B. VERIFYING THE INSTALLATION TAPE'S VALIDITY

If you encounter inexplicable problems shortly after you install Medley, they may be due to files being corrupted — the release tape may have been damaged, errors may have occurred while the tape was being read, etc. If you have unexplained problems, we recommend that you verify the checksums of your installed files.

The script generates checksum files named `*.check` and compares them to the released `*.sum` residing in the `/checksumdir` subdirectory.

The checksum script reports inconsistent files, the correct checksum values for the files, and an error message. The checksum of individual files can be generated with the UNIX command `sum` filename.

`ldechecksum [-cg]` *medleydir [dir | dirgroup]* **[Command]**

-c Generates checksums for your installed files and compares them with correct values. This is the default action.

-g Generates checksums for the files specified.

medleydir Name of the Medley installation directory. Default is **`/usr/share/lde`**.

dir Any specific directory residing under *medleydir*. Only relative pathnames with respect to *medleydir* are accepted.

dirgroup The directory group, either **all** (the default) or **lisp**, which includes the **`X/install.xxxx`**, **`X/lisplibrary`** and **`X/lispsysouts`** directories.

Output

As it begins checking each directory, the script prints a message in the form:

`Checking directory:` **`/usr/share/lde/`***subdir*

Error and warning messages may be in one of two forms:

`< E > 32711 49 4045XLPSTREAM.DFASL`

indicates that file `4045XLPSTREAM.DFASL` is erroneous or does not exist in the directory. The correct checksum of 32711, together with the size (49 Kbytes) of the file, are shown.

`< W > /usr/share/lde/fonts/display/chinese : Directory not installed`

indicates that Chinese fonts were not installed or were removed after Medley was installed.

Examples

`prompt%` **`ldechecksum /usr/share/lde`**

All files in the installed Medley directories in **`/usr/share/lde`** are checked.

```
prompt% ldechecksum /usr/share/somedir/lde lisp
```

This example checks all files in:

```
/usr/share/somedir/lde/install.xxxx
/usr/share/somedir/lde/lisplibrary
/usr/share/somedir/lde/lispsysouts
```

```
prompt% cd/usr/share/lde
```

```
prompt% ldechecksum  -c . fonts/display
```

This example checks only the display font directories. The period (**.**) is used because you are positioned under the current Medley installation directory.

[This page intentionally left blank]

APPENDIX C. LAYOUT OF INSTALLATION TAPE FILES

Layout of Installation Tape

Below follows the layout of the Medley Installation Tape with a description of the individual files.

FILE 1	FILE 2	- - - - - -	FILE 6

File	Contents	Description
1	`./install-medley`	The Medley installation utility
2	`./medley`	The Medley startup script

3	`./install.sunos3/` (only on the Sun3 installation tape)
	`./install.sunos4/`
	`./install.sunos4.1/`	

Each subdirectory contains:

`lde`	Used as a bootstrapper to load the right emulator, depending on the frame-buffer of your host and whether X Windows is running.
`ldeether`	Used when you want to use the XNS protocol from within Medley on a Sun. It will set up your system to intercept XNS and PUP packets and then immediately runs **`lde`**.
`ldesingle`	The emulator used to run Medley on a workstation with a monochrome display or one with a color frame-buffer of type cg2, cg4, or cg9.
`ldemulti`	The emulator used to run Medley on a workstation with a color frame-buffer of type cg3 or cg6.
`ldex`	The emulator used to run Medley on a workstation where an X Windows server is running.

`ldesingle.o`
`ldemulti.o`
`ldex.o`
	These object files are used when recompiling the emulators to either include your own C subroutines or when problems arise.

`makefile` `usersubrs.c`	Used when you wish to link your own C subroutines into the emulator (a non-documented feature).
`ldeether.c`	The source code for the **`ldeether`**. Its only purpose is to allow you to recompile the ethernet set-up code should you run into any problems.

4	`./lisplibrary`	Contains all the Medley 2.0 Lisp Library files
5	`./checksumdir`	Contains `ldechecksum`, `checksum` and ***X*.sum** checksum files (See Appendix B for a detailed explanation)
	`./lispsysouts`	Contains the sysout, lisp.sysout
6	`./fonts/display`	Contains the display fonts (See Table C-1 for a detailed description of the individual font files)
	`./fonts/interpress`	Contains the Interpress printer fonts (See Table C-1 for a detailed description of the individual font files)

Font Directories

Table C-1 shows the organization of the font directories, as well as the descriptions and contents of the directories.

Table C-1. Font Directories

Directory Name	Description	Font Families	Font Types
./fonts/display/presentation ./fonts/interpress/presentation	All presentation fonts for display and user interface applications	Helvetica Gacha Times Roman	Sans serif Monospace screen font in 8, 10, 12 MRR Serif
./fonts/display/publishing ./fonts/interpress/publishing	All publishing fonts for character sets, foreign characters, and technical alphabets	Classic Modern Terminal	Serif; in all character sets, sizes, faces sans serif; in all character sets, faces, but with selected sizes Monospaced, in all character sets, faces, but with selected sizes
./fonts/display/printwheel ./fonts/interpress/printwheel	All printwheel fonts for word processing applications	BoldPS LetterGothic Titan	Proportional serif Monospaced sans serif Monospaced serif
./fonts/display/JIS1 ./fonts/interpress/JIS1	Japanese Kanji fonts, character set 1	Classic	Point sizes 8 through 24
./fonts/display/JIS2 ./fonts/interpress/JIS2	Japanese Kanji fonts, character set 2	Classic	Point sizes 8 through 24
./fonts/display/chinese ./fonts/interpress/chinese	Chinese character fonts	Classic Modern	Point sizes 12 and 24 12 point
./fonts/display/miscellaneous ./fonts/interpress/miscellaneous	Miscellaneous fonts for nonstandard and rare applications	ClassicThin Hippo Logo Math OldEnglish Symbol Tonto	Brackets and parentheses in point sizes 16, 20, 26, and 30 Greek or Latin Xerox logo Math symbols Point sizes 10 and 18 Math symbols Thick monospaced 14 point MRR
./fonts/press	All metric information for Press printers.		

Manually Extracting Files from the Installation Tape

You can manually extract individual files or directories from the Medley installation tape. For example, if you want to extract the X-window emulator **ldex** for SunOS release 4.1 from the tape do the following:

```
prompt% mt -f /dev/nrst0 rewind
```

Ensures that the tape is positioned at the beginning of the tape.

```
prompt% mt -f /dev/nrst0 fsf 2
```

Positions the tape at the beginning of the third file on the tape. The **n** in the **/dev/nrst0** makes sure the tape is not rewound after the command has been completed.

```
prompt% tar xvf /dev/nrst0 ./install.sunos4.1/ldex
```

Extracts **ldex** from the Medley installation tape and puts it in your current working directory.

APPENDIX D. DIFFERENCES BETWEEN XEROX WORKSTATIONS AND THE UNIX VERSION OF MEDLEY

Local Disk and Floppy Functions

The functions for controlling device-specific behavior of the Xerox 1100 series workstation disk drives are not supported. These functions signal the error

```
Wrong machinetype
```

if called when running under UNIX. These functions include

```
IL:PURGEDSKDIRECTORY
IL:CREATEDSKDIRECTORY
IL:VOLUMESIZE
IL:DISKFREEPAGES
IL:DISKPARTITION
IL:SCAVENGEDSKDIRECTORY
IL:FILENAMEFROMID
```

The following functions for controlling the Xerox 1100 series workstation floppy disk drive also signal an error under UNIX:

```
IL:FLOPPY.FORMAT, IL:FLOPPY.NAME, IL:FLOPPY.TO.FILE,
IL:FLOPPY.FROM.FILE, IL:FLOPPY.ARCHIVE, IL:FLOPPY.UNARCHIVE,
IL:FLOPPY.MODE, IL:FLOPPY.FREE.PAGES, IL:FLOPPY.CAN.READP,
IL:FLOPPY.CAN.WRITEP, IL:FLOPPY.WAIT.FOR.FLOPPY,
IL:FLOPPY.SCAVENGE
```

These functions signal the error

```
Floppy:  No floppy drive on this machine. Device error:
{FLOPPY}
```

The following functions have no effect and always return NIL on UNIX:

```
IL:VOLUMES
IL:LISPDIRECTORYP
IL:DSKDISPLAY
```

Library Modules Not Supported on the Sun

The following modules listed in the manual *Lisp Library Modules,* Medley Release, are not supported on the Sun Workstation running Medley.

TCP, TCPCHAT, etc.

Because SunOS supports TCP/IP directly, TCP packets cannot be routed to Medley. For this reason, the TCP library modules are not supported on the Sun Workstation.

DLRS232C, DLTTY

The DLRS232C and DLTTY library modules are specific to the hardware devices available on the Xerox 1100 series workstations. Serial lines and other devices can be accessed from Medley either through sub-shells, or by using the `{UNIX}` file device, e.g., writing to `{UNIX}/dev/ttya` or `{UNIX}/dev/ttyb`.

The following library modules are normally used with equipment attached to the Xerox 1186 RS232 serial lines:

FX-80DRIVER
4045XLPSTREAM
KERMIT
RS232CHAT.

KEYBOARDEDITOR, VIRTUALKEYBOARD

Medley does not include versions of KEYBOARDEDITOR or VIRTUALKEYBOARD library modules that know about the Sun keyboards.

VIRTUALKEYBOARD lets you bring up keyboard images that give you access to special characters via the mouse. The keyboard itself is unaffected.

[This page intentionally left blank]

MEMORANDUM

FROM: John Sybalsky

DATE: September 15, 1991

RE: Release 2.0 of Medley for the Sun Workstation

Enclosed is the software and documentation for Release 2.0 of Medley for the Sun Workstation. The package consists of the following:

- Tape containing the revised software.
- Release Notes, providing warnings and information important to the successful running of the software, followed by fixed bugs.
- *Medley for the Sun Workstation User's Guide*, encompassing release contents, instructions for installing Release 2.0, and information on using it. This *Guide* has been completely reorganized, and information about using the new installation script has been added.
- *Lisp Library Modules* revised pages, reflecting additions to the prior issue (replace the old sections with the corresponding new pages).

GLOSSARY

access permissions* Determines what operations can be performed on a file.

alias* A user-created C-Shell command defined in terms of other commands or programs. For example, if you type (or put in your `.cshrc` file)

```
alias runlde "lde ~/sysout -k xx"
```

then when you type runlde to the C-Shell, it acts as if you had typed

```
lde ~/sysout -k xx
```

backing store A Xerox 1100 series workstation file, the virtual memory partition. This file stores pages as they are allocated or flushed from real memory.

byte code emulator A byte-code instruction interpreter. Executes the Interlisp-D virtual machine instruction set compatibly with microcode for the Xerox workstations.

chmod* A program used to change access permissions of a file.

chown* A program used to change ownership of a file.

{DSK} A host device name allowing users to access the SunOS file system. Uses conventions (e.g., version numbers and file name recognition which ignores the case of letters) similar to those used by the Xerox 1100 series workstation local disk device ({DSK}).

environment variable* A name/value pair that is passed to subprocesses. Can be set from the shell with the setenv command. By convention, environment variable names use uppercase rather than lowercase letters, e.g., LDEDESTSYSOUT. The Medley environment variables are LDESRCESYSOUT, LDEDESTSYSOUT, LDEINIT, LDESHELL.

home directory* The working directory when a user logs in.

host access key A special code which must be entered to Medley to run Medley software on the Sun Workstation.

lde Lisp development environment.

ldeether A program produced during the software startup procedure; runs `lde` after enabling access to Xerox network protocol.

.login* The name of a file in the home directory that is read by the shell when a user first logs in. Contains C-Shell commands.

Medley The Venue programming environment; also, the name of the release. Supports Common Lisp and Interlisp; a library of utilities, graphics packages, applications; a complete windowing system; network protocols. Runs on both Xerox and Sun workstations.

NFS* Network File System; the way SunOS handles remote file systems.

pathnames* In UNIX, a position identifier of a file or directory within the file system tree structure.

An *absolute* pathname gives the position, beginning with the root directory, of the file or directory in the file system hierarchy. Each directory in the pathname is delimited by a slash (/).

A *relative* pathname locates the position of the desired file or directory from the working directory. Again, all directories in this pathname are delimited by the slash (/).

root directory* The root of the directory tree. Designated by a slash (/) at the *beginning* of an absolute pathname. Slashes elsewhere in a pathname are simply delimiters.

shell* Command interpreter (akin to the Medley Exec).

shell script* A file that contains shell commands. Can be run by typing the file name provided the user has execute permission on the file.

site initialization file A Lisp file, used when Medley is started up. Contains standardized information about the site environment such as pointers to fonts and site parameters.

SunOS Sun's version of UNIX.

suntools A Sun system window–based program tool. A program that allows all of the Sun window–based tools to run on the screen.

tar A program for copying data to and from magnetic tape.

{UNIX} A host device name allowing users to access the SunOS file system using UNIX naming conventions. Files on the {UNIX} device have no version numbers and file name recognition distinguishes between upper– and lowercase letters.

*** Indicates a UNIX term. See UNIX documentation for full definition.**

[This page intentionally left blank]

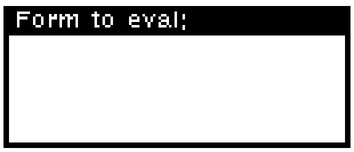
INDEX

== IMINDEX ==

The file IMINDEX contains the functions used for creating and editing index image objects, and inserting them into a Tedit document. When a Tedit document containing index objects is formatted for printing, the index objects do not appear, but information about the index objects is put into an auxilliary "IMPTR" file. The functions in IMTOOLS can be used to take a set of IMPTR files, and generate an index.

Adding an Index Object to a Tedit Document

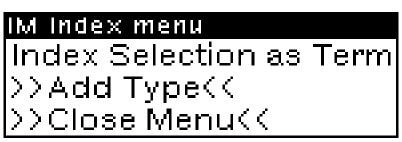
The simplest way of adding an index object to a Tedit document is to type ctrl-O while typing at Tedit, which will cause a window to pop up asking you to type a form to eval:



Typing (IM.INDEX.CREATEOBJ) will create an empty index object, indexing the term NIL, inserted in the Tedit document at the caret. Index objects appear in the Tedit window as words with boxes around them:

Using the IM Index Menu

An easier way to put many index objects into a Tedit document is to type (IM.INDEX.MENU) at the lisp exec, which will prompt you to position a menu that looks like this:



If the caret is blinking in a Tedit window, selecting [Index Selection as Term] with the left button will create and insert an index object that indexes the selected string. For example:

Before: 

, and after: 

Selecting [Index Selection as Term] with the MIDDLE button will insert an index object that indexes the selected string, and then create an index object editing window, that can be used to edit the values of the fields in an index image object.

Editing an Index Object in a Tedit Document

If you select an index object, a menu will appear asking whether you want to edit the contents of the index object:

If you select [Edit Index], you will be prompted to position an index object editing window, which looks like the following:

This is a freemenu that allows you to edit the values of the fields in an index image object. Selecting [Store Props] stores the values in the editor into the object itself -- if the "Name:" field is changed, the Tedit window will change to show the new index name. Selecting [Close Window] will close the editor window. Note that any changes are lost if you close an index object editing window without storing the new property values.

Index Object Properties

There are more properties in an index object than show in the window initially; the window can be scrolled or reshaped to see and edit the other properties. The fields are interpreted as follows:

Note: The properties listed with "()", such as "Type():", interpret their value as a list of take a list of items, delimited by spaces. The other properties interpret their value as an atom, including spaces.

Name: Value is the name used to sort and merge index entries. This should normally be all-uppercase. The [Index Selection as Term] item in the IM Index menu will automatically uppercase the selection if it is not all-uppercase, and put the real value in the "Text:" field. Examples:
Name: FOO
Name: BAR

Type(): Value is the type of object being indexed. If NIL, this stands for an English term. Other types are used for indexing lisp functions, variables, etc. Note that upper/lower case is important. Examples:
Type(): Function
Type(): Editor Command

Note: The special types CHAPTER and SUBSEC (all-uppercase) are used to create entries in the table of contents, as described below. These cannot be used as index entry types.

Text: Value is the name actually printed in the index. If NIL, the value of the "Name:" property is used. This is usually used when indexing non-uppercase items. Another place where this is useful is when you want an item containing strange characters to appear in a different place in the index. For example, if Name: = FOO and Text: = *FOO*, the item "*FOO*" will appear in the index amoung the "F" items instead of the "*" items.

Info(): Value is a list of "information" words, that mean something to the indexing programs. There are a few info words likely to be used by IMINDEX users. One is the word *PRIMARY*, which indicates that this is a primary reference. In the index, primary index page numbers are printed first, in a bold font. Another is the word *NOPAGE*, which indicates an index entry that should not be printed with a page number. This can be used to generate entries such as "FOO, see BAR". Example: Info(): *PRIMARY*

SubSec(): Probably not much use for IMINDEX users. Value is a reverse list of nested subsection and chapter numbers. For example,
SubSec(): 4 2 99
indicates subsection 4 inside subsection 2 inside chapter 99. This is not used with normal index entries, except that chapter numbers, if given, are used when generating the page number in the index. For example, if the index entry specified chapter 99, and it was on page 5, it would appear in the index as 99.5.

Another use for this field is if you use IM index objects to generate a table of contents, by creating index objects with type of CHAPTER or SUBSEC, as described below. In this case, the SubSec field is used to specify the subsection numbers.

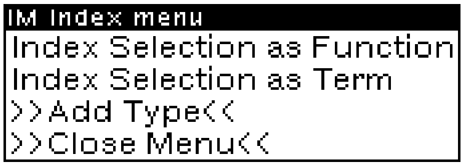
Page#: The value of this field is replaced with the page number when the Tedit textstream containing the index image object is hardcopied. Not much use to change, but it is possible.

SubName:
SubType():
SubText: Name, Type, and Text values for a sub-entry, if any, which appears under the main entry in the index.

SubSubName:
SubSubType():
SubSubText: Name, Type, and Text values for a sub-sub-entry, if any.

Adding New Types to the IM Index Menu

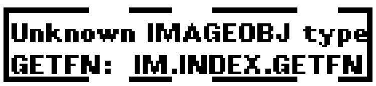
If all of the index entries in a document are terms, it is very easy to use the IM Index menu to index them. However, if there are a lot of Functions, variables, etc., you may need to edit each index entry to change the type. To make this easier, you can use the [>>Add Type<<] button in the IM Index menu to add new selections to the IM Index menu. If [>>Add Type<<] is selected, the system will prompt you to type a type name, and add a new item to the menu. For example, if you started with the initial menu, and added a new entry for the type "Function", the menu would be changed to look like the following:



If [Index Selection as Term] is selected, an index object will be created indexing the the current Tedit selection, with the type of Function.

Saving and Retrieving Tedit Documents with IM Index Objects

Tedit documents containing IM Index objects can be saved using the ordinary Tedit "Put" command, which will store all of the indexing information in the index objects. Note, however, that it is necessary to load the IMINDEX package before editing any Tedit documents containing IM Index objects -- otherwise Tedit will print the message "WARNING: Document contains unknown image objects", and all index objects will appear as the following:



If this happens, it will not be possible to use these index image objects, or re-save the Tedit document. Close the Tedit window, load IMINDEX.DCOM, and call Tedit to edit the document again.

Hardcopying a Tedit Document Containing Index Objects

A Tedit document containing index objects can be hardcopied using the normal [Hardcopy] menu button, or from the Filebrowser. Index image objects will not appear in the final hardcopy. However, while the document is being formatted, the index objects will put indexing information in an auxilliary file with the extension "IMPTR". The first time an index object is formatted, it will try to open an imptr file on the connected directory, printing a message in the prompt window: "Opening index pointer file: {DSK}<LISPFILES>FOO.IMPTR...done". When the formatting is completed, another message will appear: "Closing index pointer file: {DSK}<LISPFILES>FOO.IMPTR;1...done".

The directory used for the imptr file is the currently-connected directory (which can be changed using the CONN command). The file name used if the file name of the Tedit file, if there is one. For example, if the Tedit file is XYZ.TEDIT, the imptr file would be XYZ.IMPTR. If the Tedit text stream doesn't have a name (for instance, if the Tedit window has been brought up but never saved) the file name NONAME.IMPTR is used. The extension is always IMPTR.

For Hackers Only: If there is a need to use a different filename for the imptr file, it is possible to specify this programmatically by changing textstream properties of the Tedit textstream before the hardcopy operation. If the value of the textstream property IM.INDEX.PTRFILENAME is non-NIL, it is used as the file name for the IMPTR file. If the value of the textstream property IM.INDEX.PTRFILE is non-NIL, it is the stream used for the IMPTR information.

Creating Indicies and Tables of Contents From IMPTR Files

The file IMTOOLS.DCOM contain the routines used to create indecies and tables of contents for documents in IM format. These will also produce indicies and tables of contents from IMPTR files created by IM index objects. Loading this file will automatically load IMINDEX.DCOM, if it is not already loaded. Note, however, that you can load IMINDEX (for the purpose of editing Tedit documents with IM index objects) without loading IMTOOLS.

IMTOOLS contains the following useful functions:

(MAKE.IM.INDEX OUTFILE.FLG VOLUME.INFO IMPTR.FILES IMPTR.TYPES) [Function]

MAKE.IM.INDEX takes a number of IMPTR files, and produces an index, formatted like the one in the IRM. If OUTFILE.FLG is NIL, the output file is just sent to the default printer. If OUTFILE.FLG is T, the outfile textstream is simply returned. If OUTFILE.FLG = anything else, it is taken as a file name of an interpress file which is created <but not printed>.

VOLUME.INFO is a list of lists specifying which chapters are associated with which volumes in a large multi-volume document (like the IRM). For an example of the format, see the variable IM.MANUAL.VOLUMES in IMTOOLS. If NIL, no volume numbers are printed in the index.

IMPTR.FILES is a list of IMPTR files to be used to create this index. If NIL, any index info loaded in by explicitly calling GRAB.IMPTR is used (see IMTEDIT.TEDIT). When doing the IRM, all of the IMPTR files are loaded into global variables using GRAB.IMPTR, but users of IMINDEX probably want to specify the values specifically.

IMPTR.TYPES should be a list of IM index types to be included in the index, specified as lists. For example, if IMPTR.TYPES = ((Function)(Variable)), only function and variable entries would be listed in the index. If IMPTR.TYPES = NIL, all index entries in the IMPTR files are used.

(MAKE.IM.TOC OUTFILE.FLG CHAPTER.NUMBERS IMPTR.FILES) [Function]

MAKE.IM.TOC takes a number of IMPTR files, and produces a table of contents, formatted like the one in the IRM. All im index entries whose type is CHAPTER or SUBSEC are used as chapter and subsection pointers.

OUTFILE.FLG and IMPTR.FILES are interpreted similar to MAKE.IM.INDEX.

CHAPTER.NUMBERS is either: NIL, meaning to generate TOC of ALL data in the specified imptr files; a single number, meaning to generate a chapter TOC for that chapter; or a list of numbers, meaning to generate a TOC for those chapters.

author: Michael Sannella
file: {Phylum}<LispUsers>IMNAME. (& .DCOM)
loads file: {Phylum}<LispUsers>HASH.DCOM
loads file: {Phylum}<LispUsers>IMTRAN.DCOM [loaded by IMNAME.UPDATE.HASHFILE]

****** The IMNAME Database Package *****

IM format is the text formatting language that the Interlisp Reference Manual is represented in. It is somewhat like TEX source code, in that there are keywords, and brackets are used to delimit text. However, IM format was specifically designed for representing the Interlisp Manual, so the "text objects" used are semantically meaningful objects within the manual, such as "function definition", "lisp code", "subsection".

IMNAME is a package designed to help people who may frequently need to modify IM format files. Usually, a large document (such as the Interlisp Manual) is stored in a number of seperate files, and it is difficult to know which file contains a particular piece of information. IMNAME contains functions for analyzing a set of IM format files and building a database of "IM Names" (functions, variables, property names, etc) with pointers to the files where they are defined. Using this database, other functions allow the user to specify an IM name, and bring up a Tedit window on the appropriate file, with the cursor positioned at the right place. This tool has been very helpful to the people updating the Interlisp Reference Manual.

****** Using IMNAME to access IM format files. *****

The normal way of using an IMNAME database is by creating an "IM name inspector". This can be done using either of the following two functions:

(MAKE.IM.INSPECTOR hashFileName)
or
(IMNAME hashFileName)

hashFileName should be the name of the IMNAME database hashfile to use for this inspector. One can create multiple IM name inspectors point to the same hashfile, or to different hashfiles. Currently, I know of only two IMNAME hashfiles; the one for the Interlisp Manual and one for the Loops Manual. If hashFileName is the atom INTERLISP or LOOPS, this Inspector will point to the appropriate hashfile. If hashFileName is ommitted, it will assume that the Interlisp Manual hashfile is desired for people whose default host is Phylum, and the Loops Manual for people on Ivy. (Other people can set up their own default IMNAME hashfile by setting the global variable IM.NAME.DEFAULT.HASHFILE).

MAKE.IM.INSPECTOR sets up an "IM Inspector Window", which contains a menu. Initially, this contains the single selection "Type an IM name", which (when buttoned) prompts the user to type a name which will be looked up in the database. If an IM name is found in the hashfile, another window will appear below the first one, containing all of the "types" that the given name is known by. For example, a name may be known as both a variable and a function. If one of these types is selected, a third menu will apear below the second, listing references to the name in different files. Selecting one of the references will move the cursor in a TEDIT window (if there exists an active TEDIT window to the appropriate file), or create a new TEDIT window to the file. [Note: If a name is only known to be of one type, the "type menu" step is ommitted.]

Other functions:

(INSPECT.IM imname hashFileName)

This allows the searching for a single IM name, using several pop-up menus. In general, it is easier to use the IM name inspector.

(GET.IM.NAME.LIST hashFileName)
Returns a list of all of the IM names known within the specified hashFileName.

****** Updating an IMNAME database. *****

IMNAME works by referencing a hashfile database containing IM names and pointers to files where these names are referenced. As a set of files is modified, this information will become obsolete. In particular, simply adding a few characters to a file will invalidate all pointers to positons later in the file. This can be tolorated for awhile (it is rare that IM format files will be changed enough such that the pointers are totally off), but eventually, it is necessary to update the IMNAME database, using the following function:

(IMNAME.UPDATE.HASHFILE oldHashFileName addFileList deleteFileList flushDisappearedFlg)

This function updates an IMNAME database hashfile. First, it looks at the list of files referenced in the hashfile, and determines which of these files have been updated, by comparing version numbers. Next, every updated file is reanalyzed with IMTRAN, and updated index information is stored into an in-core hasharray. Finally, the entries in the old hashfile are read in, merged with the new info, and written out to a new hashfile.

oldHashFileName is the name of the hashfile to be updated (Note that this file must be named explicitly --- no file searches are done so that the user will not inadvertantly start updating the main manual database). The new hashfile will be created as the new version of the same file name. addFileList is a list of files that will be analyzed, and added to the database. deleteFileList is a list of files that will be deleted from the database. addFileList and deleteFileList can be used to "manage" a database, as new files are added to a document, and old ones are removed, split up,or renamed. flushDisappearedFlg determines what IMNAME.UPDATE.HASHFILE will do if it finds that some of the files in the database have disappeared (and they are not named on deleteFileList). If flushDisappearedFlg = T, the info for those files will simply be deleted. If flushDisappearedFlg = ERROR, IMNAME.UPDATE.HASHFILE will return without doing anything if files have disappeared. If flushDisappearedFlg = <anything else>, the info on the disappeared files will simply be retained.

To create a new IMNAME hashfile, pass a non-existant file name as oldHashFileName, and give a list of files as addFileList. In this case, a new hashfile will be created just from the internal hasharray info.

****** The IMTEDIT IM-to-TEDIT translation program. ******

author: Michael Sannella
file: IMTEDIT.DCOM
loads file: IMTRAN.DCOM
related files: IMTOOLS.DCOM

IM format is the text formatting language that the Interlisp Reference Manual is represented in. It is somewhat like TEX source code, in that there are keywords, and brackets are used to delimit text. However, IM format was specifically designed for representing the Interlisp Manual, so the "text objects" used are semantically meaningful objects within the manual, such as "function definition", "lisp code", "subsection". IM format is described in detail below.

IM format files are easy to edit using Tedit, but they don't look very pretty. To produce the manual, use the function IM.TEDIT, which translates IM format files to formatted Tedit text streams. These text streams can be proofread and edited by the user, or automatically printed.

A useful feature of IM.TEDIT is that is very forgiving about errors in IM format, even misplaced bracket errors! This is not to say that the output will be pretty, but at least the translation program will not bomb out on you.

***** Formatting a File with IM.TEDIT *****

To translate an IM format file into a formatted Tedit text stream, use the following function:

(IM.TEDIT *INFILE.NAME OUTFILE.FLG***)** [Function]

This function takes an IM format file, and produces a formatted Tedit text stream. *INFILE.NAME* is the name of an IM format file. *OUTFILE.FLG* determines what happens to the translated textstream. If *OUTFILE.FLG* = T, the Tedit textstream is returned by IM.TEDIT. A Tedit window showing the translated document can be created by typing **(TEDIT (IM.TEDIT** *xxx* **T))**. If *OUTFILE.FLG* = NIL, the document is immediately sent to the printer. If *OUTFILE.FLG* = anything else, it is taken as a file name for the Interpress file which is created <but not printed>.

As IM.TEDIT runs, it prints out warning messages. These messages are also saved in the file <infile>.IMERR.

Important note: It is necessary to understand that IM.TEDIT produces a totally separate document from the IM format original. Any edits to this document will NOT be reflected in the original. In general, all of the editing should be done to the IM format files, to insure that there are no inconsistancies.

Note: If IM.TEDIT is called with *OUTFILE.FLG* = T to produce a textstream, and **IM.INDEX.FILE.FLG** (below) is set to T to add index information to the textstream, the textstream cannot be stored or printed. **IM.INDEX.FILE.FLG** should only be set to T if IM.TEDIT is called with *OUTFILE.FLG* not T, so it automatically prints the document to a printer or interpress file.

***** Variables Affecting IM.TEDIT *****

The operation of IM.TEDIT affected by a number of variables:

IM.NOTE.FLG [Variable]
If T, notes will be printed out, otherwise they will be suppressed. Initially NIL. (Note: If this is T, the translation programs will print out a message to remind you that notes will be printed.)

IM.DRAFT.FLG [Variable]
If T, the output will have "--DRAFT--" and the date printed on the top and bottom of every page. Initially NIL.

IM.CHECK.DEFS [Variable]
If T, checks whether variables and functions are bound/defined in the current Interlisp environment, and prints a warning if not. For functions, will also check arg list consistancy. Initially NIL.

IM.EVEN.FLG [Variable]
If T, IM.TEDIT will add an extra page at the end of the file saying "[This page intentionally left blank]". This can be used if you need a document with an even number of pages (for double-sided copying). Initially NIL.

The following FLGs are only of interest when generating an index:

IM.INDEX.FILE.FLG [Variable]
If T, index information will be added to the formatted Tedit text stream, and the file <infile>.IMPTR will be generated containing index information when the formatted Tedit textstream is printed. Initially NIL.

IM.REF.FLG [Variable]
If T, the translation program will try to resolve cross-references by looking at various hash tables. Initially NIL.

IM.SEND.IMPLICIT [Variable]
If T, send "implicit references" for functions and variables (if not in index hash array). Initially NIL. {fn...} or {var...} objects generate "implicit references" if they are not contained in the index hash tables. This can be used to find variables and functions that are not formally defined, but only mentioned.

***** Producing an Index and Table of Contents *****

The file IMTOOLS.DCOM contains functions for gathering index information, generating an index and a table of contents, and using index information to resolve cross-references.

Currently, these tools are rather primitive. Eventually, it is hoped that they will be improved to make this process easier.

To produce an index and TOC:

(1) Evaluate (SETQ IM.INDEX.FILE.FLG T), which tells IM.TEDIT to create index pointer files <XX>.IMPTR.

(2) Run (IM.TEDIT <file> <ipfile>) on all files in the document

(3) Evaluate (INIT.INDEX.VARS) to clear the index.

(4) For each of the files in the document, load the index information by evaluating (GRAB.IMPTR xxx.IMPTR).

(5) Generate an index by evaluating (MAKE.IM.INDEX OUTFILE.FLG), where the argument OUTFILE.FLG is interpreted the same as in IM.TEDIT.

(6) Generate a table of contents by evaluating (MAKE.IM.TOC OUTFILE.FLG), where the argument OUTFILE.FLG is interpreted the same as in IM.TEDIT.

To generate formatted files with cross-references resolved, do the following:

(1) Evaluate (SETQ IM.INDEX.FILE.FLG T), which tells IM.TEDIT to create index pointer files <XX>.IMPTR.

(2) Run (IM.TEDIT <file> '{NULL}FOO.IP) on all files in the document

(3) Evaluate (INIT.INDEX.VARS) to clear the index.

(4) For each of the files in the document, load the index information by evaluating (GRAB.IMPTR xxx.IMPTR).

(5) Evaluate (SETQ IM.REF.FLG T), which tells IM.TEDIT to resolve cross references using the loaded index information.

(6) Format all of the files using (IM.TEDIT <file> <ipfile>).

Note that creating a formatted file with cross-references requires formatting the file twice.

The process of formatting a large set of documents such as the IRM can automated using the following function:

(DO.MANUAL *CHAPNAMES MAKE.INDEX.FLG GET.REFS NO.IP.FLG***)**   [Function]

Formats the IRM manual chapters specified by CHAPNAMES, producing Interpress files, "imptr" files (explained below), and error files. Uses the global variable **IM.MANUAL.DIRECTORY** (initially {erinyes}<lispmanual>) to indicate where the IRM files are taken from, and where the processed files should go.

Note: **DO.MANUAL** initially puts all files on {DSK}, and then copies them to the value of **IM.MANUAL.DIRECTORY**. Therefore, you should have at least 3000-4000 free pages on DSK.

"IMPTR" files are files with the extension "**IMPTR**" (such as ChapLitatoms.IMPTR), which contain index info, including the page number where each index appears. These are generated whenever a chapter is formatted. These must be read in to generate an index, or format a chapter resolving cross-references.

CHAPNAMES indicates which chapters should be processed. If *CHAPNAMES* = a list of chapter file names such as (ChapLitatoms ChapStack), only these chapters are formatted. If *CHAPNAMES*=**T**, all chapters in the IRM are formatted. If *CHAPNAMES*=**NIL**, all chapters that have been modified since the corresponding Interpress files were made are formatted. The global variable **IM.MANUAL.CHAPTERS** specifies all of the chapters, and all of the sub-files in each chapter, and the chapter numbers.

If *MAKE.INDEX.FLG* is non-**NIL**, after processing the chapters, the index, title pages, and many tables of contents are generated. Note that if this is done, all of the IMPTR files for all of the chapters are loaded again.

If *GET.REFS* is non-**NIL**, all of the IMPTR files are loaded before any of the chapters are processed, and all cross-references in the chapters are resolved.

If *NO.IP.FLG* is non-**NIL**, the chapters are formatted to the interpress file **{NULL}FOO.IP**, and thus not kept. This can be used when you simply want to generate all of the IMPTR files, but do not want the interpress files.

Ideally, all you should need to do to re-format the IRM is:

(DO.MANUAL T NIL NIL T) --- to generate the IMPTR files

(DO.MANUAL T T T) -- to generate the Interpress files, and the index.

If **DO.MANUAL** doesn’t do exactly what you want, the code is fairly self-explainatory.

***** Modifying the dimensions of the formatted pages *****

Note on modifying the formatting tools for different paper sizes: All of the lengths that IMTEDIT uses (left margin size, page height, etc.) are stored in global variables, set in the filecoms of IMTEDIT. IMTEDITCOMS also includes comments documenting the meaning of each of these global variables. Modify these variables until you get what you want.

***** Known problems with IMTEDIT *****

Currently, IM.TEDIT produces a good-looking document. However, there are some features that I was not able to provide, because Tedit did not provide the formatting capability. These are as follows:

(1) Footnotes. Tedit does not supply footnotes. Currently, I "fake" footnotes by positioning them in-line after the paragraph wherein they were created. They also will not appear within definitions, tables, or lisp code.

(2) Tables. Tedit doesn’t do table formatting. Currently, IM.TEDIT simply prints out the items in the table without formatting.

****** IM Format Description *****

This is a complete description of the syntax and vocabulary of "IM format". The general syntax is not likely to change, but the vocabulary will probably be extended as the Interlisp Manual is edited, and we discover new text objects that we need.

IM Format Syntax
****

An IM-format file consists of a linear string of visible, editable characters (and Tedit image objects, which are treated like characters). No funny control characters are allowed. "Text" is defined to be a linear string of characters, organized into paragraphs (delimited by blank lines), interspersed with any number of "Text Objects". Text Objects are used to specify the meaning of various pieces of text, which may be processed and formatted in different ways.

Text Objects can be divided into two types: those that take a single unnamed argument, and those that take a number of labeled arguments. All of the 'arguments' to Text Objects can be arbitrary text, organized in paragraphs, and including sub-text-objects nested to any level.

Text Objects within a file and arguments within a text object are specified using the characters "{" and "}". These characters are ALWAYS interpreted as Text Object delimiters or Text Object argument delimiters---there are special text objects for indicating that you really want a left or right bracket character as part of your text.

The format of Text Objects is:

{<TOname> <TOarg>} - single argument TO
{<TOname> {<TOargname1> <TOarg1>} {<TOargname2> <TOarg2>} .... } - multi-argument TO

Between a TOname or TOargname and its TOarg, and between named TOargs, there can be any combination of spaces, tabs, and CRs. These can be used to make your file look better.

In order to help with the problem of matching brackets in large TOs (such as SubSecs, which can extend over many pages), I have introduced a new piece of syntax: Begin and End:

"{Begin <TOname> <tag>}" is treated exactly like "{<TOname> "
"{End <TOname> <tag>}" is treated exactly like "}"
except that additional checks are made when the file is processed that the "Begin"s and "End"s match up. The (optional) "<tag>"s can be used as an additional check, to distinguish between different TOs of the same type. If you use a Begin TO, you should use a matching End TO.

Note that Begin and End TOs can be used with TOs that have labeled args, for example:

{Begin SubSec <tag>}
{Title ----}
{Text
.
.
}
{End SubSec <tag>}

In order to allow the IM format translation program to recover from errors (like missing brackets), information has been included in the program about what TOs "should" appear in other TOs. In general, "complex" TOs such as LabeledLists can only appear in other complex TOs. "Simple" TOs such as Lisp can appear within anything. These rules are defined on a per-argument basis for TOs with multiple labeled arguments --- for example, a LabeledList can appear within the "Text" argument of a FnDef, but not within the "Name" argument. This feature should not cause any problems, as long as TOs are used "reasonably."

Other rules followed by the translation program: FnDef’s, VarDef’s, Def’s, etc can only appear at the "top-level", or inside a SubSec or Chapter. (this rule is very useful, because it helps trap many bracket errors before they propogate too far) Footnotes may not be nested.

New feature <which may get changed>: If a TO name is unrecognized, it will be printed out surrounded by brackets. For example, {FOO} will actually print as "{FOO}". This allows simple expressions to be bracketed without actually using the "{bracket ...}" TO.

Text Objects currently defined:
****

(note: the order of arguments in multi-argument TOs IS significant.)

(note: No distinction is made between upper and lower case in the TO names and the argument names, or in Begin/End tags)

(note: Many TO names and argument names have synonyms. These are indicated below.)

************************* Plain Text Text objects ******

<paragraph>

All plain text is organized into paragraphs, delimited by blank lines.

{Chapter {number <number>} {title <title>} {text <text>} }

Specifies the number, title, and text of a chapter. If the number is not specified, the IM format translator will ask you to supply a number.

{subsec {title <title>} {text <text>} }

This can be used to generate sections, subsections, etc. to any depth. Heirarchical numbering is done automatically.

{Comment <text>}

Used to insert comments (which won’t appear in the final formatted output).

{Note <text>}

Inserts comments that may be printed in the final formatted output, depending on the value of the variable IM.NOTE.FLG. Should be used for comments such as {Note I should write something about X here}

{foot <text>}

Generates a footnote. Footnotes may not occur within other footnotes. Currently, Tedit does not support footnotes, so these are just printed after the paragraph in which they appear.
Synonyms: footnote -> foot

{sub <text>}

Subscripts <text>.

{super <text>}

Superscripts <text>.

{it <text>}

Used to italisize pieces of text.
Synonyms: italics, emphasize -> it

{rm <text>}

Used to print text in the default, "roman" font.

{lisp <text>}

The text object for normal single-line references to lisp code. This is used for writing things like: "Obviously, {lisp (CONS ’A ’B)} evaluates to {lisp (A . B)}."

{lispcode <text>}

The text object for multiline lisp code, which do not appear in the middle of text, and have to be formatted differently. Spaces, tabs, and carriage returns are significant within <text>.

************************** TOs for Interlisp manual objects ******

{FnDef {Name <name>} {Args <args>} {Type <keywords>} {Text <text>} }

This is used to define all lisp system functions. It needs to know the name of the function, and the args, and the text of the function description. (If the function has 0 args, the {Args --} argument may be omitted.) {Type ...} is an optional argument used to specify the argument type of the function. If the keywords NLambda or NoSpread (case doesn’t matter) are included in <keywords>, the function is specified to have the corresponding argument type.
Synonyms: FnName -> Name
FnArgs -> Args
FnType -> Type

{vardef {name <name>} {text <text>} }

Used to define system variables.

{propdef {name <name>} {text <text>} }

Used to define property names.

{MacDef {Name <name>} {Args <args>} {Type <keywords>} {Text <text>} }
Like FnDef, but for macros.

{arg <name>}
Used to talk about an abstract argument to a function, such as "x" or "y" or "number". It is used primarily within function definitions.

{fn <name>}
Used to talk about the name of a function. i.e.: "...it calls {fn CONS} to do something."

{var <name>}
Used to talk about a system variable.

{prop <name>}
Used to talk about a property name.

{Mac <name>}
Like Fn, but for macros.

{EditCom <name>}
Used to talk about an edit command.

{BreakCom <name>}
Used to talk about a break command.

{PACom <name>}
Used to talk about a programmer's assistant command.

{FileCom <name>}
Used to talk about a file package command.

************************* Indexing Text objects ******

The following TOs (for indexing, defining, and referencing) all deal with objects with specific "object-types." The Interlisp Manual contains a very large number of names (CONS, NIL, FOO, etc). It is not enough just to list the name in the index --- it is also important to indicate WHAT the name is (a function, a variable, an error message, etc.) An "object-type" is essentially the description that would be printed in the index to describe a particular name. In IM format, such a description is given by a list of words, within parenthesis. [note upper/lower case are NOT distinguished in "object-types"] For example:

(Function)
(Error Message)
(Transor Command)
(Compiler Question)

Some commonly-used object-types may be abbreviated with single words:

FN	->	(Function)
Var	->	(Variable)
Prop	->	(Property Name)
BreakCom	->	(Break Command)
EditCom	->	(Editor Command)
PACom	->	(Prog. Asst. Command)
FileCom	->	(File Package Command)
Error	->	(Error Message)
Litatom	--,	
Atom	--->	(Litatom)

[note: In {PageRef ...} and {SectionRef ...}, the word TAG can be used to refer to tags, as specified below. The word FIGURE is a synonym for TAG; this can be used to refer to a figure tag. The word TERM can be used in {index ...} to index an English term.]

{index <text>}

Creates an index reference. Some of the text objects (such as function definitions) will automatically create an index reference, but it is useful to be able to create one explicitly. This should have the format:

{index <keywords> <object> <object type>}

<keywords> (optional) can be any combination of the words
BEGIN
END
PRIMARY
<object> can be any number of words.
<object type> should be an object-type as specified above.
If this is omitted (<text> does not end with ")" and the last word in <text> is not one of the special object-type words), then this is the index of a term. This can also be specified by using the word TERM.

Examples: {index *PRIMARY* BLOBBY (Transor Command)}
{index SELF-DESTRUCT SEQUENCE INITIATED Error)
{index *BEGIN* *PRIMARY* file names)

{indexX {name <name>} {type <type>} {info <info words>} {text <text>}
{subname <name>} {subtype <type>} {subtext <text>}
{subsubname <name>} {subsubtype <type>} {subsubtext <text>} }

Used for creating special index references whose printname is different from the "index name", or who have sub-index entries, or should not have page numbers in the index. <name> is the index name, used for referencing the object, and alphabetizing the index entry. <type> is the object type. <info words> (optional) can contain one or more of the keywords *BEGIN*, *END*, *PRIMARY*. <text> is the text printed in the index for this index entry.
Example: {indexX {name +} {type (Infix Operator)} {text {lisp {arg X}+{arg Y}}}} could be used to index "+" as an infix operator so that it would appear in the index alphabetized near other "+"s, but printing as "A+B".

The arguments {subname <name>} {subtype <type>} {subtext <text>} {subsubname <name>} {subsubtype <type>} {subsubtext <text>} are all optional. These are used to specify 1st and 2nd level subentries in the index. The subentry can have their own type and text definition, although these are mostly terms.

If <info terms> includes *NOPAGE*, this indicates that the index entry should not have a page number associated with it. This can be used with subentries to create "See also..." notes. Within a list of subentries, the *NOPAGE* entries will always be listed last.

Example: The following set of indexx commands:

{indexX {name FOO} {type (Big Command)} {text {lisp /FOO/}}}
{indexX {name FOO} {type (Big Command)} {subname ZZZ}}
{indexX {name FOO} {type (Big Command)} {subname BAZ}}
{indexX {index *NOPAGE*} {name FOO} {type (Big Command)} {subname See also BAR}}
{indexX {name FOO} {type (Big Command)} {subname BAZ} {subsubname QWERTY} {subsubtype VAR}}

Would produce the index entries:

/FOO/ (Big Command) x.xx
    BAZ x.xx
        QWERTY (Variable) x.xx
    ZZZ x.xx
    See also BAR

{Def {type <type>} {name <name>} {printname <pname>} {args <args>} {parens} {noparens} {text <text>} }

May be used to specify the definition of anything. <type> should be an object-type, as specified above. <name> should be the name of the object, ideally a single word. If {printname <pname>} (optional) is given, it will be used for printing the object in the top line of the definition and in the index. If {printname <pname>} is not given, the printed name is specified by {name <name>}, {args <args>}, {parens}, and {noparens}. <args> should be the "arguments" of this object. [{args <args>} is optional.] Normally, {Def will put parenthesis around the object if and only if {Args <args>} is given. {parens} and {noparens} (if given) can be used to change this default behavior. Examples:

{Def {Type (Blobby Command)} {Name DOBLOB} {Args A B}
    {Text ....}}
{Def {Type (CLISP Character)} {Name +} {PrintName {lisp {arg X}+{arg Y}}}
    {Text ....}}

{Tag <tag>}

This may be used to associate <tag>, a single word (upper/lower case doesn't matter), with a particular place in the text. This can be referenced by {SectionRef and {PageRef as described below.

{SectionRef <object-type> <object>}
{PageRef <object-type> <object>}

These TOs are used to provide a cross-reference facility. {SectionRef will print "section " and the section that the "main" occurence of the object with the given type occurs. <"main" shall remain undefined for now> {PageRef prints "page " and the appropriate page number. If <object-type> is "Tag", these TOs can refer to tags generated with {Tag <tag>}. Examples:

{SectionRef Fn CONS} could print "section 3.1.1"
{PageRef (File Package Type) Expressions} could print "page 14.34"
{PageRef Tag CompilingCLISP} could print "page 20.42"

Note: {PageRef...} and {SectionRef...} can only be used to reference 1st level index references. They cannot be used to reference subentries generated with {indexx...}

{Term <text>}

Prints <text> exactly as given, and also puts it into the index. Equivalent to "<text>{index <text> Term}".

{Figure {tag <tag>} {text <text>} {caption <text>} }

Prints a simple "figure", with figure text and caption as specified by the "text" and "caption" arguments. "{tag <tag>}" is an optional argument which, if specified, generates a tag at the beginning of the figure. This tag can be used to refer to the page or section that this figure appears on (using {PageRef Figure <tag>} or {SectionRef Figure <tag>}), or to refer to the figure number (using {FigureRef <tag>})

{FigureRef <tag>}

Prints "figure" and the figure number of the figure with the specified tag.

************************* Complex Text objects ******

{numberedlist {item <text>} {item <text>} {item <text>} ...}

Used for making numbered lists of items.

Synonyms: Text -> Item

{unnumberedlist {item <text>} {item <text>} {item <text>} ...}

Like {numberedlist ...}, except that it uses bullets to mark each item in the list.

Synonyms: UnlabeledList -> Unnumberedlist
Text -> Item

{labeledlist {name <text>} {item <text>} {name <text>} {item <text>} ...}

Used for making a table associating a set of labels with descriptions of their meanings.

The argument {LName <text>} can be used to add a label left-justified on the page. Either an LName or a Name, or both, can appear as labels for an item. If an LName is supposed to be on the same line as a Name, it should appear first: {Labeledlist {LName xx} {Name yy} {item zz} ...}

Synonyms: Label -> Name
Text -> Item

{Table {Column} ... {Column} {First <text>} {Next <text>} {Next <text>} ... {First <text>} ... }

Used for formatting multi-column tables. The number of {Column}'s before the first {First or {Next indicates how many columns the table should have. If no {Column}'s are given, the default is three columns. There are other formatting arguments, but they will probably change. <<<Note: This is not currently supported in Tedit. Currently, this TO only prints out the values of each {First} or {Next} argument, without formatting. Do not use tables>>>

************************* Other Text objects ******

{Include <filename>}

Used to include the text of one file within another. Files may be nested arbitrarily deep.

{Lbracket}
{Rbracket}
{bracket <text>}

These are used to insert the curly-bracket symbol into the text. {Lbracket} and {Rbracket} insert left and right brackets, and {bracket <text>} puts brackets around the given piece of text.

The following TOs are used to specify certain non-typeable characters.

{CRsymbol} or {cr}	-	carriage return symbol (currently a superscripted 'CR')
{pi}	-	Greek letter pi
{plusminus}	-	Plus or Minus ("+" + "-")
{GE}	-	greater than or equal (">" + "=")
{LE}	-	less than or equal ("<" + "=")
{NE}	-	not equal ("=" + "|")
{endash}	-	short dash
{emdash}	-	long dash
{ellipsis}	-	ellipsis (...)
{bullet}	-	bullet
{anonarg}	-	long dash (should be used in the argument list of a function definition to specify the "unspecified arguments" specified in the current manual with a dash)

FORMATTING AN INDEX

Creating a properly formatted index takes a certain amount of work (about 2 hours).

- Type in your Executive `(TEDIT (MAKE.IM.INDEX T NIL '(filenames) NIL))`

- Save the resulting file in {ERIS}<Doc>LoopsBeta>Ref>RefIndex-Raw.tedit.

- Now start dithering. The task is the change all the separators in the file. For example, the index program returns pages in the form 1.2; 10.5,38; 18.3-4. The Index format we use is in the form 1-2; 10-5; 10-38; 18-3. Unfortunately, you can't make all the necessary changes in a global substitute, since many of the separators are also in the text and the comma requires that you repeat the chapter number. Here goes:

  -- Fix the commas. Do a Find on a comma (,). Everyhwere a comma occurs in a page number, change it to a semicolon (;) and repeat the chapter number and a dot before the page number. For example, 10.5,38 becomes 10-5; 10-38.

  -- Fix the dashes. Do a Find on a dash (-). Everyhwere a dash occurs in a page number, delete it and the following page number.

  -- Fix the period. Except for he form #., you can use a global substitute from a period to a dash (. to -). This takes about 20-30 minutes, as there about 700 substitutions.

  I really recommend saving the file at this point.

- Hardcopy the index. Make whatever page breaks and other changes you feel are necessary. Make a final hardcopy and save the file.

ARCLEANUP Package --- Michael Sannella
(and augmented by Susana Wessling, 26-mar-87)

The file {eris}<lispcore>internal>library>ARCLEANUP.LCOM contains a few functions used to update the AR database and to print summaries.

(AR.CLEANUP UPDATE.FLG INDEX.LOCAL.DIR SUMMARY.FLG SUMMARY.LOCAL.DIR]

This is the main function, which should be run every few days, preferably just as you go out the door (so your machine can crunch on it overnight). If UPDATE.FLG is non-NIL, all of the ARs that have been touched since the last update are scanned, and the AR database is updated. If SUMMARY.FLG is non-NIL, a set of summaries is generated on {eris}<lispars>summaries> and press-fied.

INDEX.LOCAL.DIR and SUMMARY.LOCAL.DIR are used if you want the AR index and/or the summary reports to be cached on a local disk. This speeds up the prosess considerably, along with reducing ethernet load. If non-NIL, each of these should be a host/directory pair, which will be PACKed on the front of a filename to generate the caching filename. These are not just flags, with caching defaulting to {DSK}, so that I can use multiple partitions for caching. The issue is space: Sometimes the local disk does not have enough room for two copies of the AR index, or for both the txt and press version of the grang summary.

Example:

AR.CLEANUP (T NIL T NIL)

-- does cleanup with no caching

AR.CLEANUP (T {DSK} T {DSK2})

--- caches index on {DSK}, and the summaries on {DSK2}

AR.CLEANUP (T {DSK})

--- just does index update, caching on {DSK}.

Another useful function:

AR.CLEANUP.REDO.SUMMARIES: just generates summaries, does not do a new cleanup.

The AREDIT Interlisp bug database system

author: Michael Sannella
files: {eris}<lispcore>library>AREDIT.DCOM
doc: {eris}<lispcore>library>AREDIT.TEDIT
uses: All Tedit files

The file AREDIT.DCOM contains a number of tools useful for examining, editing, and submitting ARs ("Action Requests") related to the Interlisp-D system. These tools are loosely based on the "Adobe" tools in the Tajo environment. The Interlisp-D support group uses this system to keep track of the state of outstanding bug reports. There are currently over 2000 ARs in the database.

These tools can be used from any machine running Interlisp-D which can establish a leaf connection to the Phylex: file server, where the database files are currently stored, and the ERIS file server.

After loading AREDIT.DCOM, the user can create two types of windows: AR edit forms and AR Query forms.

The AR Edit Form

An AR edit form is used to examine, edit, and submit ARs. To create an AR Edit Form, evaluate (AR.FORM). Interlisp will prompt you to specify a region for the form window -- the best size to give it is one about half the width of the screen and at least half the height of the screen. The form window which will appear contains three subwindows: (1) On the top is the message subwindow, where prompts and status messages are printed; (2) in the middle is the command subwindow, a menu of commands for editing / submitting ARs; (3) on the bottom is the form subwindow, where the information in an AR is displayed.

The command subwindow contains the following commands:

New -- Buttoning this word clears the fields of the AR in the form subwindow. Some fields (**Source, Submitter, Status**) are initialized to appropriate values for a new AR.

Get -- Buttoning this retrieves the AR whose number follows "**Number:**" in the command subwindow.

Put -- Buttoning this will either store an edited AR, or submit a new AR. Which one (submit new or store old) depends on whether the last operation was "**New**" or "**Get**". If the current AR displayed was retrieved with "**Get**", then "**Put**" will store it as the old AR. If this AR was built up from scratch after buttoning "**New**", then "**Get**" will submit is as a new AR. The title of the form subwindow gives an indication of what state the form is in: if it says "New Bug Report", then "**Put**" will submit it. If it says "Editing AR xxx", then "**Put**" will store it. [There are plans to improve this interface]

Number: -- This is a text field just like text fields in the form window (see below) used to specify the number used by "**Get**". Buttoning the word "**Number:**" will pending-delete-select the value of the field, so you can delete it and insert a new number. If the character carriage-return is typed, then a "**Get**" is automatically done on the value of this field. This is faster than typing a number, and buttoning "**Get**".

The form subwindow contains a large number of fields. The meaning of these fields is described in XXX. The value of these fields can be edited as follows:

"Enumerated fields" can only contain certain values. These are indicated in the form subwindow by field names followed by curly-braces "{}". To change the value of one of these fields, button the field name; a menu of permissable values will appear; select a value; it will be inserted between the braces.

[note: Some of the enumerated field values are dependent on other fields. For example, the values of "Subsystem:" depend on the value of "System:" -- if the "System:" value is changed, the "Subsystem:" value is automatically set to NIL. The fields with this relationship are System:/Subsystem: and Machine:/Disk:.]

"Text fields" can contain arbitrary text. These fields do not have braces after the field name. The text can be edited using normal Tedit editing. Buttoning the field name will pending-delete-select the entire field value, which allows the whole field to be easily deleted.

[note: Currently, stored ARs only contain straight text. Any tedit formatting information put into an AR will be lost when the AR is stored. Image objects (like bitmaps) are also not stored.]

[note: A few of the text fields, like "Number:", are read only --- they cannot be edited by the user.]

[note: in older versions of AREDIT, some text fields (such as "Attn:" could only contain a certain number of characters. The user could type as many characters as he wanted, but an error would occur when the "Put" command was executed. This has now been changed --- any text field can contain an arbitrary number of characters.]

The AR Query Form

An AR Query Form is used to search the AR database for all ARs with particular characteristics. One can search for all ARs with a given name in the "**Attn:**" field, all ARs which have Status: = Open, etc. These ARs can be sorted, and a summary of the selected ARs can be printed into a file.

To create an AR query form, evaluate (AR.QFORM.CREATE). Interlisp will prompt for a region (the default size is ok), and create a window with three subwindows: (1) on top, a message subwindow, for printing prompts and messages; (2) in the middle, a browser subwindow, used for displaying the ARs seleced by a query; (3) on the bottom, the AR query command subwindow, containing a number of commands and fields.

The Ar Query command subwindow contains the following fields/ commands:

Query List: -- This field is used to specify which ARs the "**Query**" command will search for. This field should be filled with an AR query spec, which has one of the following forms:

(<field> HAS <val>) searches for all ARs whose text field <field> contains <val>. <string> may either be an Interlisp string or an atom. The search is case-independent: foo matches Foo matches FOO.

(<field> IS <val>) searches for all ARs whose enumerated field <field> has the value <val>.

(AND <spec1> <spec2> ... <specN>) returns all ARs satisfying ALL of the given specs.

(OR <spec1> <spec2> ... <specN>) returns all ARs satisfying ANY of the given specs.

[note: an implicit (AND) is wrapped around the value of the "Query List:" field, so just giving a number of specs will AND them together.]

Not every AR field can be searched for in the same way: some can only be searched with HAS, some can only be searched with IS, and some (like the Description: field) cannot be searched at all. To find out the possible query specs, button the words "**Query List:**" --- this will put up a menu of all of the permitted searching options. Some of these menu items have submenues. When one of the options is selected, it is added at the end of the value of this field.

Examples:

Query List: (Subject: HAS foo)
Searches for all ARs whose subject contains the string "foo".

Query List: (Status: IS Open) (Attn: HAS sannella)
Searches for all open ARs which have "sannella" in the Attn: field.

Query List: (OR (Status: IS Declined) (Status: IS Superceded))
Searches for all ARs with Status: either Declined or Superceded.

Sort List: -- This field determines how the Query-ed ARs will be sorted. Currently, ARs can only be sorted by the values of enumerated fields. Buttoning the words "**Sort List:**" will put up a menu of the permitted field names -- selecting one will add it to the value of this field.

Example:

Sort List: Status: Priority:
This will sort first by the Status: field, and then by the Priority: field.

[note: After sorting by all given fields, if two ARs are the same, they are sorted by AR number. Therefore, if this field is left blank, the queried ARs will be in numerical order]

Query -- Buttoning this command will initiate the query specified by the "**Query List:**" field, and sort it according to the value of the "**Sort List:**" field. While the query is in progress, the AR query command subwindow is greyed-out. When the query is completed, the numbers, subjects, etc of the ARs which have been found are displayed in the AR query browser subwindow. This window can be scrolled both vertically and horizontally.

Print File: -- This field can be filled in with a file name, which is used to specify the file that the **Print** command should store a report. If left blank, a window will pop up on the screen, and the information will be displayed there.

Print -- This prints a detailed summary of all of the ARs from the last **Query** into the file given in the **Print File:** field.

To generate and print a summary of a selected group of ARs, fill in the **Qury List:** and **Sort List:** fields, select **Query** and wait for the query to complete, fill in the **Print File:** field, and select **Print**. The summery is rather wide -- it may be a good idea to use the LANDPRESS package to printit out sideways on a printer.

The AR query browser window:

This window shows a short summary (one line each) of the ARs that have been queried. Left-buttoning one of the AR lines will call AR.SHOW on that AR, to display it. Middle-buttoning an AR line will "**Get**" that AR into a specified AR edit form window.

Background menu commands:

When AREDIT is loaded, the item "AREDIT" is added to the background menu, with a number of subitems. These are interpreted as follows:

AREDIT -- Creates a new AR edit form, initially cleared.

New AR Form -- Same as AREDIT

Load AR Form -- prompts the user for a number, and creates a new AR edit form, with the specified AR number loaded initially.

AR.SHOW -- prompts the user for a number, and calls AR.SHOW, an old version of AREDIT which quickly displays the contents of a given AR. It prompts for a window region the first time it is used -- thereafter it uses the same window.

AR Query Form -- Creates a new AR query form.

Auxiliary AR edit form commands.

Pressing the left mouse button in the title bar of the AR Edit form command subwindow will bring up a menu of less-used commands:

Clear -- Clears ALL the fields of the current AR. Similar to New, except that non of the fields like Source:, etc., are filled in. This is useful when you are submitting an AR for someone else.

New
Get
Put -- the same as the Commands in the edit command subwindow.

Put&Get -- prompts the user for a number, Puts the current AR, and Gets the given numbered AR. Useful when scanning through a number of ARs.

GetFromFile -- Prompts the user for a file name, and loads the information from that file into the AR edit form subwindow. If this file is not in the right format, an error will be generated.

PutToFile -- Prompts the user for a file name, and stores the information from the AR into that file.

Locally caching the AR index.

All AR query operations use the information in the "AR Index" file. This file, which is updated every few days, is stored as {eris}<lispars>AR.INDEX. To speed up Query operations, this file can be copied to a local file server, or the local hard disk. Warning: this file is currently ~700 IFS pages long, and it will undoubtedly get larger. Also, it is the responsibility of the user to make sure that they update their local version of AR.INDEX when the master copy is updated.

To use a local version of AR.INDEX, give the file name as an argument to AR.QFORM.CREATE:

(AR.QFORM.CREATE '{DSK}AR.INDEX).

Global Variables that control AREDIT

The following are global variables that can be set to alter the operation of AREDIT. These are the only global variables that it is safe to change.

AR.ENTRY.LIST.WINDOW.FIELDS -- Controls which fields are displayed in the AR query browser window, along with the widths of the fields.

AR.ENTRY.LIST.PRINT.FIELDS -- Controls the fields displayed by the Print command of the AR query form.

AR.ENTRY.LIST.PRINT.MULTILINE.FLAG -- if non-NIL, the Print command of the AR query form will print all of the characters in each field, using multiple lines for those field values bigger than the field width allowed. If NIL, each AR will use only one line, truncating any field values that are too big.

DATEPATCH

By: Bill van Melle
(vanMelle.pa@Xerox.com)

DATEPATCH fixes some bugs and extends the functionality of the date functions `DATE`, `GDATE` and `IDATE`.

Date Parsing

The date parser (`IDATE`) now handles all dates legal in RFC822 syntax (except the silly single-digit military time zones). In addition, it handles months spelled out, months abbreviated with a period, and ignores initial strings of the form "{letter}*,", assuming these to be specifying a day, as in "Monday, May 1, 1989". In addition to the official time zone specifications, it also recognizes any in the list `TIME.ZONES`, whose format has changed slightly:

`TIME.ZONES` [Variable]

An association list whose elements are of the form (*offset regzone dstzone*), where *offset* is the number of hours west of GMT (note that this, unfortunately, is opposite in sign to the RFC822 standard, but is strictly an internal matter), *regzone* is a string specifying the time zone normally, and *dstzone* is a string specifying the zone when daylight savings time is in effect. If *dstzone* is omitted, then there is no representation for that zone in daylight savings time, and `DATE` is forced to use absolute syntax (e.g., +0400).

The initial value of `TIME.ZONES` is

```
((8 "PST" "PDT")
 (7 "MST" "MDT")
 (6 "CST" "CDT")
 (5 "EST" "EDT")
 (0 "GMT" "BST")
 (0 "UT")
 (-1 "MET" "MET DST")
 (-2 "EET" "EET DST"))
```

`IDATE` also accepts an optional argument DEFAULTTIME, which is interpreted as a number of seconds past midnight. If the date string does not contain a time, DEFAULTTIME is used; if DEFAULTTIME is `NIL`, `IDATE` returns `NIL` in this case (as it always has).

Date Output

The date printers (`DATE` and `GDATE`) now produce appropriate time zones outside of the U.S. Given a choice of time zones, they take the first entry found in `TIME.ZONES`. In addition, they support a few more `DATEFORMAT` options:

`MONTH.LONG` [DateFormat Option]

Provides for full names of months rather than the first three characters. For instance, `(DATE (DATEFORMAT MONTH.LONG SPACES NO.TIME))` might produce "20 February 87".

`MONTH.LEADING` [DateFormat Option]

Causes the month to be produced as a word before the day and the day to be followed by a comma. For instance, `(DATE (DATEFORMAT MONTH.LEADING MONTH.LONG YEAR.LONG NO.TIME))` might produce "February 20, 1987". `MONTH.LEADING` implies `SPACES` and inhibits `NUMBER.OF.MONTH`.

`CIVILIAN.TIME` [DateFormat Option]

Specifies 12-hour time instead of 24-hour (military) time. For instance, `(DATE (DATEFORMAT CIVILIAN.TIME NO.DATE NO.SECONDS))` might produce "11:34pm".

For completeness, listed below are all the `DATEFORMAT` options currently supported.

Those affecting the date portion:

`NO.DATE`	Omit the date portion (month, day, year, day of week).
`NUMBER.OF.MONTH`	Use a number for the month instead of spelling it.
`MONTH.LONG`	Spell the month out instead of abbreviating it.
`MONTH.LEADING`	Month before day, spelled out, comma after day.
`YEAR.LONG`	Use 4 digits for year instead of 2.
`DAY.OF.WEEK`	Include day of week (it appears at the end of the string, in parentheses).
`DAY.SHORT`	Use 3-letter abbreviation for day.
`SLASHES`	Separate parts of date with slashes instead of hyphens.
`SPACES`	Separate parts of date with spaces instead of hyphens.
`NO.LEADING.SPACES`	Omit leading spaces (default is a fixed format that always "lines up"). This also affects the hour when `CIVILIAN.TIME` is specified.

Those affecting the time portion:

`NO.TIME`	Omit the time portion (hour, minutes, seconds, zone).
`NO.SECONDS`	Omit seconds.
`CIVILIAN.TIME`	12-hour instead of 24-hour time.
`TIME.ZONE`	Include time zone specification.

FILEBANGER

Filed as {eris}<lispcore>internal>library>FileBanger.*
Not for customer release

Basic Functions

(DOFILEBANGER *Destination Length NoBreak*)

Starts a new file banger in its own process. The device and directory specified by *Destination* will be where the test files are created during the testing. Each file will be *Length* bytes long. The file banger performs consistency checks on all the files it creates after each operation. If those consistency checks fail, it will print a message indicating what it found wrong, and if *NoBreak* is NIL will open a break window.

To stop the file banger process, one may (KILL.PROCESS 'FILEBANGER), provided that only a single file banger is in operation.

(FILEBANGER *TestFile Destination MakeWindow NoBreak InParms OutParms*)

Enters a loop of creating, manipulating, and deleting files, making consistency checks after each pass. *TestFile* may be the name of a file (whose contents are used as the contents of the test files), or an integer, in which case a test file of thet many random bytes is created. *Destination* is the device and directory to be tested (normally this should be {DSK}<LispFiles>). *MakeWindow*, if T, causes a fresh window to be created; that window is used for progress messages as the test runs. If *MakeWindow* is NIL, the regular TTY window for the process is used (and created, if need be!). If the test's consistency checks fail, FILEBANGER fill print a message in the status window; if *NoBreak* is NIL, it'll also stop and open a break window. If you are interested in really stressing the file system, you may specify PARAMETERS arguments to be given to OPENSTREAM when files are opened. *InParms* is used when opening for input, and *OutParms* when opening for output. We don't recommend the use of these arguments.

Generally speaking, the test file should be of a good length (5000 bytes or more), to assure that the files spread out to cover the disk volume you're testing.

FILEBANGER continues to run until interrupted from the outside somehow. The easiest way to arrange that is to run it in its own process (using DOFILEBANGER) and use KILL.PROCESS.

Recommended Operation

(DOFILEBANGER '{DSK}<LispFiles> 5000 NIL)

This will open a TTY window for the FILEBANGER process, and print small ongoing status messages there; you will need to do something to prevent the scrolling from hanging things up when the TTY window is full (either type ahead some spaces in that window, or arrange for the window not to pause when it fills up). To stop the file banger when it has run long enough, do (KILL.PROCESS 'FILEBANGER) in the top-level TTY window.

Likely Error Messages

HARD DISK ERROR

The low-level disk code hit a hard error on the disk, and was able to complete its operation to the extent of recognizing the error. Indicates a bad spot on the disk.

VERIFY ERROR

The disk-page label checking code found an inconsistency. While this might be software, it is most likely to be controller trouble or a bad spot on the disk.

(infinite uninterruptible loop)

The low-level disk code hit a snag so severe that it could never complete the disk operation at all. This should never happen. If it does, suspect IOP trouble.

File1 and File2 differ at byte xxx

Two files should have been identical but weren't. Suspect software problems, unless the error follows a particular drive, processor, or IOP.

File1 has length xxx, but File2 has length yyy.

Two files should have been identical but weren't. Suspect software problems, unless the error follows a particular drive, processor, or IOP

Files in this directory /usr/local/lde/i ternal/
where copied from {eris}<lispcore>internal>library>
31-Jan-90

LISP DIAGNOSTICS

Date : 05 - 21 - 1985

This document describes Lisp Diagnostics program for Xerox 1108
and
Xerox 1109 workstations.

To start it, load LISPDIAGNOSTICS.DCOM from the Lisp Library.

As soon as it finishes loading the file, a menu will appear.
You will have 2 choices, i.e. : Start Exercise and Stop Exercise.

If the file has previously been loaded, you'll only have to type :
(MAKEDIAGNOSTICSMENU) to make the menu appear.
When you select Start Exercise, the following message will appear on the Top level typescript window :

Legend

! -> Completed !DIAGNOSE of MACROTEST
@ -> Completed TANSPEED benchmark
# -> Completed BROWSE benchmark
$ -> Found a Clearing House on the Ethernet
- -> Looked, but failed to find a Clearing House
[xxx] -> Tried 32 retrievals from CH and got xxx failures
{xxx} -> Copied and deleted xxx copies of the Disk file
(xxx) -> Finished with xxx'th run of the EMUPROC loop
GDATE on new line marks release of working set pages

In addition to that, a window appears on top of the menu, it describes
the
current activity indicated by a black rectangular cursor.
They are :

1. Swap out working set.
2. Ethernet activity.
3. Disk Activity.
4. Benchmark.

To stop the diagnostics, simply click the Stop Exercise selection.

(This package was superficially tested on Intermezzo>Full.sysout on
05 - 21 - 1985 by G. Santosa)

NSMAIL

INTRODUCTION

The module NSMAIL implements the protocols to allow Lafite to be used to send and retrieve Xerox NS Mail. Load the file NSMAIL.LCOM. To run this in Lyric, you must have loaded the LispUsers module NSRANDOM as well (q.v. for important loading information). If you don't have NSRANDOM loaded, you can't use the "Put to file" command described below.

If you have both Grapevine and NSMAIL implementations loaded, you must set Lafite's mode to NS. Use the "NS Mode" subitem underneath Lafite's Quit command, or call (LAFITEMODE 'NS). You must also be a registered NS user, and have a mailbox.

ATTACHMENTS

The main difference between this and earlier versions of NSMAIL is that "attachments" are no longer left in your mailbox to be read later with, for example, Viewpoint. Instead, Lafite retrieves the entire attachment and encapsulates it into an image object that is enclosed as part of the text message, immediately following the header. A typical attachment appears in a mail message as:

Attachment: Viewpoint Document

If you click inside the object with any mouse button, you are offered a menu of things you can do with the attachment. The choices vary according to the type of attachment:

View as text — This brings up a window in which is displayed the raw content of the attachment as ascii bytes. Runs of non-ascii bytes are replaced by nulls to reduce the amount of garbage. Some attachments are utter gibberish, but some, such as Viewpoint documents and Interpress masters, contain sections that are plain text. With this command, you may be able to decide whether you care to do anything further with the attachment. (Sorry, there is no Viewpoint to TEdit converter, nor are there plans for one.)

Put to file — This prompts you for a file name, and creates a file to contain the attachment. The file must be on an NS file server for this command to be very useful; otherwise, information will be lost. Once the file is so stored, you can retrieve it from Viewpoint and manipulate it just as if you had originally retrieved it as mail in Viewpoint.

Send to Printer — This command is only available for attachments that are in the form of an Interpress master. The command prompts you for a printer (must be one that accepts Interpress, of course), and sends the attachment to it for printing.

Expand folder — This command is only available for attachments that are in the form of a "folder". A folder is a mechanism for collecting several objects into a single one. The Expand folder command splits the attachment up into its component objects, each of which can be manipulated in the same way as a top-level attachment. For example, if the folder contains an Interpress master, you can print it.

If you use the Put to file command on a folder, the name component of the file name you type will be treated as the name of a new subdirectory, and the components of the folder will appear as files in that subdirectory. For other types of attachments, Put to file (usually) produces an ordinary (non-directory) file.

Messages containing attachments are otherwise just like formatted messages—you can move them to other folders, and you can forward them (assuming the mail is received by another Lafite recipient and did not have to pass through a mail gateway).

There is currently no mechanism for creating your own attachments to send to other users.

MISCELLANY

If you prefer the old behavior of leaving the attachments behind in the mailbox, set the variable NSMAIL.LEAVE.ATTACHMENTS to T, but this use is discouraged. You must take care to regularly retrieve your mail from somewhere (such as Viewpoint) that will flush out all the mail; otherwise, the mail with attachments, whether you want them or not, accumulate on the server.

When in NS mode, Lafite will want your NS login identity. Normally, if your NS password differs from your default password, you will be prompted to login. You can also call (LOGIN '|NS::|) yourself to set your NS login.

You can freely intermix Grapevine and NS mail in the same mail folder if you like, but the Answer command always treats the selected message as if it were one in the current mode. So if you try to answer a Grapevine message while in NS mode, some confusion may result. Also, the status window always shows you mail status only of the current mode.

SOURCELOOKUP

INTRODUCTION

This module provides a mechanism to locate the source file and floppy in which a particular system function is defined. It is designed for use with the set of Xerox LISP source files distributed on floppy with the Lyric release. This module uses the WHEREIS module distributed with Lisp Library documented in Section 23 of the ***Interlisp Reference Manual***.

LOADING THE MODULE

SOURCELOOKUP requires WHEREIS.DCOM and HASH.DCOM to be loaded. The variable WHEREIS.HASH must be set to the hash file LYRICSOURCES.WHEREIS. All these files are distributed on the floppy labeled Lyric Sources #10. The hash file must reside on a random access device. To use the hash file on FLOPPY do the following:

(SETQ WHEREIS.HASH '({FLOPPY}LYRICSOURCES.WHEREIS))

The variable LyricSourceIndex will be set to a list where each element is a list whose CAR is the name of a Lyric Source floppy and CDR is the list of files contained on the floppy.

USER FUNCTION

(LOCATE.FUNCTION *function*)

If *function* is defined in a distributed source file prints out the appropriate file name and floppy and returns the file name; else prints *"function* not found" and returns NIL.

EXAMPLES

(LOCATE.FUNCTION 'SETQ)
The function **SETQ** is defined in the file **LLINTERP** located on floppy **Lyric Source #5**.
(LLINTERP)

(LOCATE.FUNCTION 'UNDEFINED)
UNDEFINED not found.
NIL

UNIXMAIL

By: Bob Bane (Bane.mv@envos.Xerox.com)

INTRODUCTION

UNIXMAIL is a new mail sending and receiving mode for Lafite. It sends mail via Unix hosts using the SMTP mail transfer protocol and can receive mail either by reading a Unix mail spool file or by calling the Berkeley mail program.

INSTALLATION

Turn Lafite off, load the file UNIXMAIL, make sure UNIXMAIL is configured appropriately (check the settings of the variables below, and make sure any other modules UNIXMAIL m

SETUP instructions for
PC Emulation virtual hard disk and real PC floppy disk access

1. Make sure all of the normal PCE .DCOM files are loaded to run PCE: (PCEWINDOW, PCE, PCEERD, PCEDISPLAY, PCEKEYBOARD).

2. Select PC Emulation on the background MENU. This may take a while to load the PC fonts.

3. Put MSDOS disk in floppy drive. Select BOOT in the command window.

4. If the PCE boots ok, then select QUIT in the the command window.

5. Now we will make a virtual hard disk for the PCE.

From LISP do the following:

```
1_(PCE.CREATE.ERDFILE 'PCE.DISK 50)
                  (* this will make a lisp file PCE.DISK which is nearly 2
                      megabytes big - make sure you have enough disk space)
```

6. Reboot the PCE (BOOT command) and make sure configuration has the right file name PCE.DISK for the ERDFILE using the CONFIG command.

7. From the PC window:

```
-FDISK
                  (* create DOS partition using all of the disk )

-FORMAT C:/V/S
                  (* format the rigid disk, put a system on it, and label)

-COPY A:*.* C:
                  (* copies all of the system files from floppy to ERD)
```

8. Select QUIT from the COMMAND window.

9. You may now boot from the ERD (emulated rigid disk) by selecting FIXED DISK as boot device and Set-Config from the configuration window.

10. From LISP exec:

```
2_ LOAD(VPCDISK.DCOM)
                  (* loads in LISP virtual PC disk/floppy and real PC floppy
access)
3_ (VPCDISK.CREATE.DEVICE 'PCDISK 'PCE.DISK T)
                  (* associates ERDFILE PCE.DISK with a device name PCDISK)
4_ DIR {PCDISK}PCDISK:*
                  (* directory listing of the PC DISK)
5_(SETQ PC.TEXTFILE.EXTENSIONS '(TXT PAS BAS 1 2 3]
                  (* list of PC file extensions which are treated as TEXT type
files)
6_(VPCDISK.CREATE.DEVICE 'PCFLOPPY '{PCFLOPPY})
                  (* associates real PC floppy driver with {PCDISK} device
PCFLOPPY)
                  (* make sure real floppy is in the disk drive when you
execute this)
7_DIR {PCDISK}PCFLOPPY:*
                  (* directory listing of the real floppy)
```

11. At this point you are set up to do any LISP commands you want such as Filebrowser, Hardcopy, TEdit, etc. using the {PCDISK}PCDISK: for the virtual hard disk, and {PCDISK}PCFLOPPY: for real PC floppies.

12. Try using Filebrowser and then selecting a textfile that has one of the extension names you setup in PC.TEXTFILE.EXTENSIONS above, and selecting HARDCOPY.

13. Try using TEDIT on a floppy file such as {PCDISK}PCFLOPPY:README.1 and then editing the file and writing (PUT) it back. and then reselect it from filebrowser and selecting hardcopy. (Make sure floppy is not write-protected.)

14. You may also access files on sub-directories. IE, {PCDISK}PCDISK:<DIR>FILE - However, you cannot delete or create directories from LISP. You have to do this from the PC window.

SOURCES>

* LOOPS 10183 10346 10456
* LOADLOOPS 10346
* MEDLEY-PATCH 10346
* SEDIT-PATCH 10189
* LOOPS-FILEPKG 10495
* LOOPSBROWSE 10235
* INSPECT-PATCH 10357
* LOOPSUTILITY 10370

LIBRARY>

* LOOPSMS 10281
* MASTERSCOPE 10346 10436
* MSPARSE 10346
* MSANALYZE 10346
* MSCOMMON 10346

USERS>

* LOOPSBACKWARDS
* CONVERSION-AIDS

USERS>RULES>

* LOOPSRULES
* LOOPSRULES-ROOT
* LOOPSRULESC
* LOOPSRULESD
* LOOPSRULESP
* LOOPSRULESTTY

Ron:
10464 Comment removal AR
10436 MASTERSCOPE fix to MSANALYZE for CLASSES, into Medley

Subject: New Lispcore>Library package: MesaTypes
To: Lispcore^, Lispsupport, Sheil, Cooper, Purcell

Announcing a new Lispcore>Library package: MesaTypes

By Tayloe Stansbury with help from Richard Burton.

This package introduces three new clisprecordtypes which allow you to describe any block of bits with an arbitrarily nested datatype. You can define multidimensional, nonstandardly indexed arrays with multi-word elements; records with multi-word fields; and multi-word types. Special accessfns cover up the necessary \BLTs and LOCFs. Appropriate create methods are automatically provided. The package also provides a number of macros for manipulating instances of such types.

Anyone who wants to use graceful datatypes to describe some arbitrary chunk of memory (e.g. ethernet, rs232, nsfiling, any file system) will probably find this package useful; most of the 1108 file system now depends on it. People translating Mesa system code into Lisp will find it particularly useful.

This message is stored as Lispcore>Library>MesaTypes.tedit. Proper external documentation of this package will follow release of the 1108 file system. Extensive examples of its use can be found in the first few pages of stansbury>newdlionfs>dlionfs.

-- Tayloe.

Common Lisp package code and symbol conversions
Ron Fischer

[This document is for developers only. It is not intended to be part of any external software release.]

The package code can behave mighty strangely if not operated "just so." Please read this document carefully.

Loading

The package code is now loaded in the full.sysout. It lives in `{Eris}<LispCore>Library>` on

```
CMLSYMBOL.DCOM
CMLPACKAGE.DCOM
```

Initializing

Packages, as you might expect, must be bootstrapped carefully. At its simplest, initialization is accomplished by calling just one function:

`(package-init t)`

simple startup

`(package-init &optional (convert? nil))` [Function]
Clear, make structures of, initialize & convert symbols (if `convert?` is `t`) to, and enable use of the symbol package system. On a DandeTiger this takes about 25 minutes.

The bootstrap actually takes place in three steps. The first one creates all the package structure needed. The second passes over all the atoms in the sysout to "tag" them with packages. The third enables the package world.

step 1: make system packages

`(package-make)` [Function]
Create, but do not fill with symbols, the base packages that need to exist. Also enables the package qualifier characters in the readtables and saves the old definitions of \READ.SYMBOL and \MKATOM.

step 2: convert existing symbols

`(package-hierarchy-init &optional (convert? nil))` [Function]
Fill all the initial system packages with their proper symbols, moving litatoms into appropriate places and such. If `convert?` is non-nil then symbols whose pnames have fake package qualifiers, like cl:length, will be converted IN PLACE to remove the qualifier. If conversion takes place you cannot fully disable the package system.

step 3: enable package system, remove old hacks

The third part of the bootstrap enables the use of packages by the reader, printer and friends. This involves redefining `\MKATOM` and `\READ.SYMBOL`. Also, if `*package*` is set to `nil` things will not go well. Please use `package-enable` and `package-disable` to start and stop, as these fellows know what they're doing. These faithful functions are availible on a menu by calling package-menu.

`(package-enable &optional (package *interlisp-package*))` [Function]
Turn on the package system, making package the current one and redefining `\READ.SYMBOL` and `\MKATOM` appropriatly.

`(package-disable)` [Function]
Turn off the package system and restore the old definitions of `\READ.SYMBOL` and `\MKATOM`. After disabling, symbols interned under the package system will not be `eq` to symbols of the same name reread.

`(package-menu)` [Function]
Make a menu that allows turning the package system on or off without using the reader.

`(package-hacks-disable)` [Function]
Eliminates package simulation hacks when loading over an old sysout. These hacks cannot be re-enabled.

`(pkgconvert-enable)` [Function]
Enables a change in the behavior of the function `\\READ.SYMBOL` (using the function `check-symbol-namestring`), which converts symbols being read based on their "looking like" a pacakge prefixed symbol name. "Looking like" is defined by a table in the global `litatom-package-conversion-table`.

`litatom-package-conversion-table` [Variable]
a global variable containing a list of clauses used by the functions package-hierarchy-init and check-symbol-namestring to determine if a symbol "looks like" is trying to be package qualified. The list contains clauses with the following structure:

(*prefix-string list-of-exception-strings package-name-string where*)

prefix-string - string containing the prefix of symbols which this clause converts, eg `"CL:"`

list-of-exception-strings - list of strings naming symbols which this clause should leave alone, eg `("CL:FLG")`

package-name-string - string containing the name of the package that symbols converted by this clause should wind up in, eg `"LISP"`

where - either `:INTERNAL` or `:EXTERNAL`, indicates whether symbols converted by this clause should be external or internal in their package.

The initial value of this variable is:

```
(      ("CL:"      ("CL:FLG" "CL:MAKE-SYMBOL" "CL:COPY-SYMBOL" "CL:INTERN"
                    "CL:MAKE-KEYWORD" "CL:GENTEMP" "CL:KEYWORDP")
                         "LISP"            :external)
       (":"        nil    "KEYWORD"    :external)
)
```

Notes & cautions

The read functions may return strings if you use package qualifier syntax and `*package*` does not contain a package. This is part of debugging code that Bill put into them. Beware especially of making `*package*` `NIL` without calling `package-disable`.

The list of conversion clauses is searched linearly, hence longer prefixes should come before shorter ones with the same chars in them, ie put a clause for `"CL::"` before one for `"CL:"`.

Missing features

Cannot be placed early into loadup due to dependancies on CL files.

Apropos (and other Interlisp litatom functions) are not redefined to operate with packages.

Performance

There have been some small improvements since these timings were taken.

```
FILEDATES : (("21-Jul-86 14:12:17" .
{ERIS}<LISPCORE>CML>LAB>CMLPACKAGES.;129))
```

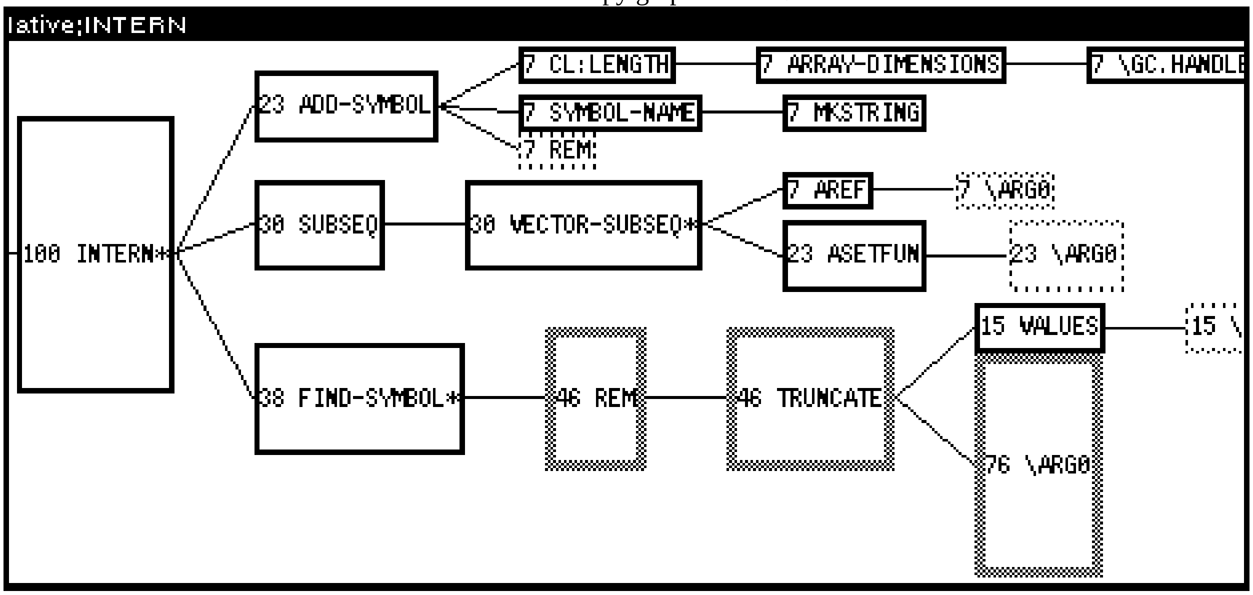
Testing symbol / litatom creation. Old array package. There are 6 random symbols made (new ones). intern is a factor of 16 slower, conses heavily, makes 3 strings for each call.

```
85#(TIMEALL (TIME.MKATOM))
Elapsed Time =       .336 seconds
SWAP time =           .32 seconds
CPU Time =           .016 seconds
PAGEFAULTS = 8
LISTP
9

86#(TIMEALL (TIME.INTERN))
Elapsed Time =       .449 seconds
SWAP time =          .188 seconds
```

```
CPU Time =           .261 seconds
PAGEFAULTS = 5
LISTP   STRINGP
423     18
```

Breakdown spy graph follows:



Peano

Maintained By: Frank Shih (Shih.envos@Xerox.com)

This document last edited on 8-Nov-88

INTRODUCTION

Peano is an ancient graphics demo.

OPERATION

(PEANODEMO *LEVEL SCALE*) [Function]

Runs a demo in PEANOWINDOW (set first time), drawing Peano curves to level *LEVEL* with lines *SCALE* units wide.

Files in this directory /usr/local/lde/internal/
where copied from {eris}<lispcore>internal>library>
31-Jan-90

Read and Print state profile

There are a large number of special variables that control reading and printing. Taken together these comprise a "mode of operation" for the reader and printer. Because there are a fair number of such variables a method for capturing and restoring their state has been provided.

(xcl:make-read-print-profile &key (readtable *readtable*) (read-base *read-base*) (read-suppress *read-suppress*) (package *package*) (read-default-float-format *read-default-float-format*) (print-escape *print-escape*) (print-pretty *print-pretty*) (print-circle *print-circle*) (print-base *print-base*) (print-radix *print-radix*) (print-case *print-case*) (print-gensym *print-gensym*) (print-level *print-level*) (print-length *print-length*))
[Function]

Creates a read-print-profile object. The default values of its components are taken from the current special bindings of the variables shown in the argument list above.

(xcl:copy-read-print-profile profile) [Function]

Creates a new read-print-profile object and copies the values in the slots of profile into the new one.

(xcl:read-print-profile-p object) [Function]

Returns true if the object is a read-print-profile, otherwise false.

(xcl:read-print-profile-readtable profile) [Function]
(xcl:read-print-profile-read-base profile) [Function]
(xcl:read-print-profile-read-suppress profile) [Function]
(xcl:read-print-profile-package profile) [Function]
(xcl:read-print-profile-read-default-float-format profile) [Function]
(xcl:read-print-profile-print-escape profile) [Function]
(xcl:read-print-profile-print-pretty profile) [Function]
(xcl:read-print-profile-print-circle profile) [Function]
(xcl:read-print-profile-print-base profile) [Function]
(xcl:read-print-profile-print-radix profile) [Function]
(xcl:read-print-profile-print-case profile) [Function]
(xcl:read-print-profile-print-gensym profile) [Function]
(xcl:read-print-profile-print-level profile) [Function]
(xcl:read-print-profile-print-length profile) [Function]
(xcl:read-print-profile-print-array profile) [Function]
(xcl:read-print-profile-print-structure profile) [Function]

profile must be a read-print-profile object. Returns the named slot of the read-print-profile. A corresponding setf method is provided for each slot.

(xcl:save-read-print-profile profile) [Function]

Capture bindings of special read & print variables into the profile. Returns profile.

(xcl:with-read-print-profile profile-form &body forms) [Macro]

Binds all the special read & print variables to the values in the profile and executes the body forms as in a let. Returns the value of the last of the forms.

(xcl:restore-read-print-profile profile) [Function]

Restore values of special read & print bindings from profile. Sets current bindings. Returns T.

xcl:*default-read-print-profile* [Variable]

Holds a simple default read-print-profile. When possible programs should default to the current settings of the read-print variables by capturing them with save-read-print-profile instead.

(xcl:find-read-print-profile name) [Function]

Since read-print-profiles enclose readtables and packages, which are availible by name, named read-print-profiles are also availible.

This function will retrieve one of the following "standard" read-print-profiles:

LISP: LISP readtable and USER package, others nominal.
XCL: XCL readtable and XCL-USER package, ditto above.
INTERLISP: INTERLISP readtable and INTERLISP package, ditto.

This function is case insensitive. It returns NIL if a profile by that name is not found.

(xcl:list-all-read-print-profile-names) [Function]

Returns a list of strings containing the names of all availible read-print-profiles.

WHEREIS

The WHEREIS package extends the function WHEREIS such that, when asked about a given name as a function, WHEREIS will consult not only the commands of files that have been noticed by the file packages, but also one or more hash file data bases that associate function names with file names.

(WHEREIS *NAME TYPE FILES FN*) [Function]

Behaves exactly like the standard WHEREIS function (see the *Interlisp-D Reference Manual*) unless *TYPE*=FNS (or NIL) and *FILES*=T. In this case, WHEREIS will consult, in addition to the files on FILELST, the hash files that are on the value of WHEREIS.HASH. Note: normally the user will just enter the full name of a file onto WHEREIS.HASH, but as WHEREIS begins to use it, it will convert the entry into a cons of the name and the hash file handle.

Many system functions, such as the editors, call WHEREIS with *FILES*=T, so loading this package automatically makes any information contained in the WHEREIS data base files available throughout the system.

Creating a Where Is Data Base

Information may be added to an existing WHEREIS hash file, or by creating new data bases. It is often useful to have a separate data base for large user systems, explicitly calling the following function:

(WHEREISNOTICE *FILEGROUP NEWFLG DATABASEFILE*) [Function]

Inserts the information about all of the functions on the files in *FILEGROUP* into the WHEREIS data base contained on *DATABASEFILE*. If *DATABASEFILE* is NIL, the first entry on WHEREIS.HASH is used.

FILEGROUP may be simply a list of files, in which case each file thereon is handled directly; but it may also be a pattern to be given as a file group argument to DIRECTORY, so * may be used.

If *NEWFLG* is non-NIL, a new version of *DATABASEFILE* will be created containing the data base for the functions specified in *FILEGROUP*. If *NEWFLG* is a number, the hash file will be created with *NEWFLG* entries. Otherwise, it will be created to allow 20,000 entries.

Example:

The following sequence of actions will cause all of the files on the PROJECT directory to be noticed by the WHEREIS package.

```
(WHEREISNOTICE'<PROJECT>*. T '<PROJECT>PROJECTWHEREIS.HASH)
    (push WHEREIS.HASH
    (FINDFILE '<PROJECT>PROJECTWHEREIS.HASH))
```

library/CLIPBOARD

Written by Ron Kaplan, 2020-2021

A small package that implements copy and paste to the system clipboard.

It arms meta-C for copy to the clipboard from the current selection of an application that has been armed (Tedit, Sedit), and also meta-X for extraction (copy followed by delete).

Meta-V is defined as an interrupt character that pastes the current clipboard contents into whatever process curent has input focus.

The information in the clipboard can be provided from or provided to external (non-Medley) applications (mail, emacs, etc.) in the usual way. For example, a form cselected in SEDIT can be copied to the clipboard and pasted into an email message.

It assumes that the external format of the clipboard is determined by (SYSTEM-EXTERNALFORMAT, and characters will be converted to and from the Medley internal character encoding.

The name of the clipboard stream may differ from platform to platform. On the Mac, the paste stream is "pbpaste" and the copy stream is "pbcopy". Those names are used if "darwin" is a substring of (UNIX-GETENV "ostype"). Otherwise both stream-names default to "xclip". The functions CLIPBOARD-COPY-STREAM and CLIPBOARD-PASTE-STREAM perform this selection.

COLOR

Introduction

This document describes software for driving color displays. In order to run COLOR, you need either a Sun (3 or 4) with CG4 color hardware and display, a Dorado (Xerox 1132) with attached color display, or a Dandelion (Xerox 1108) with attached BusMaster and color display.

The color software which is distributed among a number of files can be divided into a machine independent group of files that all users can usefully load and a machine dependent group containing files that work for particular combinations of hardware.

The machine independent color graphics code is stored in the library files LLCOLOR.LCOM and COLOR.LCOM. LOADing COLOR.LCOM causes LLCOLOR.LCOM to be LOADed.

The machine dependent portions of Xerox Lisp color software is stored in files such as MAIKOCOLOR.LCOM, DORADOCOLOR.LCOM, or COLORNNCC.LCOM. The user LOADs one of these files according to what kind of machine and color card the user is using.

The Sun color driver resides in the file MAIKOCOLOR.LCOM which loads LLCOLOR.LCOM and COLOR.LCOM. The CG4 device suppports 8 bpp at 1152 by 900 resolution. The user must be running ldecolor, the special color capable emulator. The physical display monitor is shared by both the monochrome and color screens (described below) .

The Dorado color driver resides in the file DORADOCOLOR.LCOM which loads LLCOLOR.LCOM and COLOR.LCOM. The Dorado color board supports four or eight bits per pixel (bpp) at 640 by 480 resolution. (The board supports 24 bpp also, but Xerox Lisp doesn't yet.)

The Dandelion color drivers reside in the files DANDELIONUFO.LCOM, DANDELIONUFO4096.LCOM, and COLORNNCC.LCOM, one package for each of three different kinds of boards. The user should load one of these packages on a Dandelion attached to a BusMaster and color display. The DANDELIONUFO and DANDELIONUFO4096 packages drive 4 bpp at 640 by 400 resolution color boards used inside Xerox which have been made obsolete by COLORNNCC. The COLORNNCC package drives an 8 bpp color at 512 by 480 resolution board, the Revolution 512 x 8, made by Number Nine Computer Corporation. The Revolution 512 x 8 is available both inside and outside Xerox through Number Nine.

Hardware Displays and Software Screens

On some workstations (such as the Dorado and Dandelion), there may be physically two separate displays. On most Suns, there is a single physical display, which additionally may be shared by two Unix devices. One device is monochrome (b/w), and the other is color.

To support the various hardware configurations and external display devices, the software has a special datatype, a "screen". There are two distinct instances of screens, a b/w screen, and a color screen. A screen represents and controls a physical hardware display, and contains windows, icons, and tracks the mouse.

On workstations with physically two separate hardware displays, each display is represented by a corresponding screen data structure. On workstations with a single hardware display, the display is shared by both the b/w screen and the color screen.

In all cases, before initialization only the b/w screen (and thus display) is visible and active. After initialization both screens are active (can contain screen images), although on single displays, only one screen is visible at a time. Since each screen logically controls a display, we will henceforth use the terms "screen" and "display" interchangeably. Screens are discussed in greater detail below.

Turning the Color Display Software On and Off

The color display software can be turned on and off. While the color display software is on, the memory used for the color display screen bitmap is locked down, and a small amount of processing time is used to drive the color display.

(COLORDISPLAYP) [Function]

returns T if the color display is on; otherwise it returns NIL.

(COLORDISPLAY*ONOFF TYPE***)** [Function]

turns off the color display if *ONOFF* is 'OFF. If *ONOFF* is 'ON, it turns on the color display allocating memory for the color screen bitmap. *TYPE* should be one of 'MAIKOCOLOR, 'DORADOCOLOR, 'DANDELIONUFO, 'DANDELIONUFO4096, or 'COLORNNCC. The usual sequence of events for the user is to LOAD the software needed to drive a particular color card and then to call COLORDISPLAY with the appropriate *TYPE* to turn the software on. For example,

(LOAD 'COLOR.LCOM)

(LOAD 'COLORNNCC.LCOM)

(COLORDISPLAY 'ON 'REV512X8)

will turn on the software needed to drive the Number Nine Computer Corporation's Revolution 512 x 8 card with 1108 and BusMaster.

Besides initializing or reinitializing a color card that has been powered off, COLORDISPLAY allocates memory for the color screen bitmap. Turning on the color display requires allocating and locking down the memory necessary to hold the color display screen bitmap. Turning off the color display frees this memory.

Colors

The number of bits per pixel determines the number of different colors that can be displayed at one time. When there are 4 bpp, 16 colors can be displayed at once. When there are 8 bpp, 256 colors can be displayed at once. A table called a *color map* determines what color actually appears for each pixel value. A color map gives the color in terms of how much of the three primary colors (red, green, and blue) is displayed on the screen for each possible pixel value.

A color can be represented as a number, an atom, or a triple of numbers. Colors are ultimately given their final interpretation into how much red, blue, and green they represent through a color map. A color map maps a color number ([0 . . . 2^{nbits}-1]) into the intensities of the three color guns (primary colors red, green, and blue). Each entry in the color map has eight bits for each of the primary colors, allowing 256 levels per primary or 2^{24} possible colors (not all of which are distinct to the human eye). Within Xerox Lisp programs, colors can be manipulated as numbers, red-green-blue triples, names, or hue-lightness-saturation triples. Any function that takes a color accepts any of the different representations.

If a number is given, it is the color number used in the operation. It must be valid for the color bitmap used in the operation. (Since all of the routines that use a color need to determine its number, it is fastest to use numbers for colors. COLORNUMBERP, described below, provides a way to translate into numbers from the other representations.)

Red Green Blue Triples

A red green blue (RGB) triple is a list of three numbers between 0 and 255. The first element gives the intensity for red, the second for green, and the third for blue. When an RGB triple is used, the current color map is searched to find the color with the correct intensities. If none is found, an error is generated. (That is, no attempt is made by the system to assign color numbers to intensities automatically.) An example of an RGB triple is (255 255 255), which gives the color white.

RGB [Record]

is a record that is defined as (RED GREEN BLUE); it can be used to manipulate RGB triples.

COLORNAMES [Association list]

maps names into colors. The CDR of the color name's entry is used as the color corresponding to the color name. This can be any of the other representations. (Note: It can even be another color name. Loops in the name space such as would be caused by putting '(RED . CRIMSON) and '(CRIMSON . RED) on COLORNAMES are *not* checked for by the system.) Some color names are available in the initial system and are intended to allow color programs written by different users to coexist. These are:

Name	RGB	Number in default color maps	
BLACK	(0 0 0)	15	255
BLUE	(0 0 255)	14	252
GREEN	(0 255 0)	13	227
CYAN	(0 255 255)	12	224
RED	(255 0 0)	3	31
MAGENTA	(255 0 255)	2	28
YELLOW	(255 255 0)	1	3
WHITE	(255 255 255)	0	0

Hue Lightness Saturation Triples

A hue lightness saturation triple is a list of three numbers. The first number (HUE) is an integer between 0 and 355 and indicates a position in degrees on a color wheel (blue at 0, red at 120, and green at 240). The second (LIGHTNESS) is a real number between zero and one that indicates how much total intensity is in the color. The third (SATURATION) is a real number between zero and one that indicates how disparate the three primary levels are.

HLS [Record]

is a record defined as (HUE LIGHTNESS SATURATION); it is provided to manipulate HLS triples. Example: the color blue is represented in HLS notation by (0 .5 1.0).

(COLORNUMBERP *COLOR BITSPERPIXEL NOERRFLG***)** [Function]

returns the color number (offset into the screen color map) of *COLOR*. *COLOR* is one of the following:

· A positive number less than the maximum number of colors,

· A color name,

· AN RGB triple, or

· An HLS triple.

If *COLOR* is one of the above and is found in the screen color map, its color number in the screen color map is returned. If not, an error is generated unless NOERRFLG is non-NIL, in which case NIL is returned.

(RGBP *X***)** [Function]

returns *X* if *X* is an RGB triple; NIL otherwise.

(HLSP *X***)** [Function]

returns *X* if *X* is an HLS triple; NIL otherwise.

Color Maps

The screen color map holds the information about what color is displayed on the color screen for each pixel value in the color screen bitmap. The values in the current screen color map may be changed, and this change is reflected in the colors displayed at the next vertical retrace (approximately 1/30 of a second). The color map can be changed to obtain dramatic effects.

(SCREENCOLORMAP *NEWCOLORMAP***)** [Function]

reads and sets the color map that is used by the color display. If *NEWCOLORMAP* is non-NIL, it should be a color map, and **SCREENCOLORMAP** sets the system color map to be that color map. The value returned is the value of the screen color map before **SCREENCOLORMAP** was called. If *NEWCOLORMAP* is NIL, the current screen color map is returned without change.

(CMYCOLORMAP CYANBITS MAGENTABITS YELLOWBITS BITSPERPIXEL**)** [Function]

Returns a color map that assumes the BITSPERPIXEL bits are to be treated as three separate color planes with CYANBITS bits being in the cyan plane, MAGENTABITS bits being in the magenta plane, and YELLOWBITS bits being in the yellow plane. Within each plane, the colors are uniformly distributed over the intensity range 0 to 255. White is 0 and black is 255.

(RGBCOLORMAP REDBITS GREENBITS BLUEBITS BITSPERPIXEL**)** [Function]

Returns a color map that assumes the BITSPERPIXEL bits are to be treated as three separate color planes with REDBITS bits being in the red plane, GREENBITS bits being in the green plane, and BLUEBITS bits being in the blue plane. Within each plane, the colors are uniformly distributed over the intensity range 0 to 255. White is 255 and black is 0.

(GRAYCOLORMAP BITSPERPIXEL**)** [Function]

Returns a color map containing shades of gray. White is 0 and black is 255.

(COLORMAPCREATE *INTENSITIES BITSPERPIXEL***)** [Function]

creates a color map for a screen that has *BITSPERPIXEL* bits per pixel. If *BITSPERPIXEL* is NIL, the number of bits per pixel is taken from the current color display setting. *INTENSITIES* specifies the initial colors that should be in the map. If *INTENSITIES* is not NIL, it should be a list of color specifications other than color numbers, e.g., the list of RGB triples returned by the function INTENSITIESFROMCOLOR MAP.

(INTENSITIESFROMCOLORMAP *COLORMAP***)** [Function]

returns a list of the intensity levels of *COLORMAP* (default is (SCREENCOLORMAP)) in a form accepted by COLORMAPCREATE. This list can be written on file and thus provides a way of saving color map specifications.

(COLORMAPCOPY *COLORMAP BITSPERPIXEL***)** [Function]

returns a color map that contains the same color intensities as *COLORMAP* if *COLORMAP* is a color map. Otherwise, it returns a color map with default color values.

(MAPOFACOLOR *PRIMARIES***)** [Function]

returns a color map that is different shades of one or more of the primary colors. For example, (MAPOFACOLOR ’(RED GREEN BLUE)) gives a color map of different shades of gray; (MAPOFACOLOR ’RED) gives different shades of red.

Changing Color Maps

The following functions are provided to access and change the intensity levels in a color map.

(SETCOLORINTENSITY *COLORMAP COLORNUMBER COLORSPEC***)** [Function]

sets the primary intensities of color number *COLORNUMBER* in the color map *COLORMAP* to the ones specified by *COLORSPEC*. *COLORSPEC* can be either an RGB triple, an HLS triple, or a color name. The value returned is NIL.

(COLORLEVEL *COLORMAP COLORNUMBER PRIMARY NEWLEVEL***)** [Function]

sets and reads the intensity level of the primary color *PRIMARY* (RED, GREEN, or BLUE) for the color number *COLORNUMBER* in the color map *COLORMAP*. If *NEWLEVEL* is a number between 0 and 255, it is set. The previous value of the intensity of *PRIMARY* is returned.

(ADJUSTCOLORMAP *PRIMARY DELTA COLORMAP***)** [Function]

adds *DELTA* to the intensity of the *PRIMARY* color value (RED, GREEN, or BLUE) for every color number in *COLORMAP*.

(ROTATECOLORMAP *STARTCOLOR THRUCOLOR***)** [Function]

rotates a sequence of colors in the SCREENCOLORMAP. The rotation moves the intensity values of color number *STARTCOLOR* into color number *STARTCOLOR*+1, the intensity values of color number *STARTCOLOR*+1 into color number *STARTCOLOR*+2, etc., and *THRUCOLOR's* values into *STARTCOLOR*.

(EDITCOLORMAP *VAR NOQFLG***)** [Function]

allows interactive editing of a color map. If *VAR* is an atom whose value is a color map, its value is edited. Otherwise a new color map is created and edited. The color map being edited is made the screen color map while the editing takes place so that its effects can be observed. The edited color map is returned as the value. If *NOQFLG* is NIL and the color display is on, you are asked if you want a test pattern of colors. A yes response causes the function SHOWCOLORTESTPATTERN to be called, which displays a test pattern with blocks of each of the possible colors.

You are prompted for the location of a color control window to be placed on the black-and-white display. This window allows the value of any of the colors to be changed. The number of the color being edited is in the upper left part of the window. Six bars are displayed. The right three bars give the color intensities for the three primary colors of the current color number. The left three bars give the value of the color's Hue, Lightness, and Saturation parameters. These levels can be changed by positioning the mouse cursor in one of the bars and pressing the left mouse button. While the left button is down, the value of that parameter tracks the *Y* position of the cursor. When the left button is released, the color tracking stops. The color being edited is changed by pressing the middle mouse button while the cursor is in the interior of the edit window. This brings up a menu of color numbers. Selecting one sets the current color to the selected color.

The color being edited can also be changed by selecting the menu item "PickPt." This switches the cursor onto the color screen and allows you to select a point from the color screen. It then edits the color of the selected point.

To stop the editing, move the cursor into the title of the editing window and press the middle button. This brings up a menu. Select Stop to quit.

Color Bitmaps

A color bitmap is actually a bitmap that has more than one bit per pixel. To test whether a bitmap is a color bitmap, the function BITSPERPIXEL can be used.

(BITSPERPIXEL *BITMAP***)** [Function]

returns the bits per pixel of *BITMAP*; if this does not equal one, *BITMAP* is a color bitmap.

In multiple-bit-per-pixel bitmaps, the bits that represent a pixel are stored contiguously. BITMAPCREATE is passed a *BITSPERPIXEL* argument to create multiple-bit-per-pixel bitmaps.

(BITMAPCREATE *WIDTH HEIGHT BITSPERPIXEL***)** [Function]

creates a color bitmap that is *WIDTH* pixels wide by *HEIGHT* pixels high allowing *BITSPERPIXEL* bits per pixel. Currently any value of *BITSPERPIXEL* except one, four, eight, or NIL (defaults to one) causes an error.

A four-bit-per-pixel color screen bitmap uses approximately 76K words of storage, and an eight-bit-per-pixel one uses approximately 153K words. There is only one such bitmap. The following function provides access to it.

(COLORSCREENBITMAP) [Function]

returns the bitmap that is being or will be displayed on the color display. This is NIL if the color display has never been turned on (see COLORDISPLAY below).

Screens, Screenpositions, and Screenregions

In addition to positions and regions, the user needs to be aware of screens, screenpositions, and screenregions in the presence of multiple screens.

Screens

SCREEN [Datatype]

There are generally two screen datatype instances in existence when working with color. This is because the user is attached to two displays, a black and white display and a color display.

(MAINSCREEN) [Function]

returns the screen datatype instance that represents the black and white screen. This will be something like {SCREEN}#74,24740.

(COLORSCREEN) [Function]

returns the screen datatype instance that represents the color screen. Screens appear as part of screenpositions and screenregions, serving as the extra information needed to make clear whether a particular position or region should be viewed as lying on the black and white display or the color display.

(SCREENBITMAP *SCREEN***)** [Function]

returns the bitmap destination of *SCREEN*. If *SCREEN*=NIL, returns the black and white screen bitmap.

Screenpositions

SCREENPOSITION [Record]

Somewhat like a position, a screenposition denotes a point in an X,Y coordinate system on a particular screen. Screenpositions have been defined according to the following record declaration:

(RECORD SCREENPOSITION (SCREEN . POSITION)

(SUBRECORD POSITION))

A SCREENPOSITION is an instance of a record with fields XCOORD, YCOORD, and SCREEN and is manipulated with the standard record package facilities. For example, (create SCREENPOSITION XCOORD _ 10 YCOORD _ 20 SCREEN _ (COLORSCREEN)) creates a screenposition representing the point (10,20) on the color display. The user can extract the position of a screenposition by fetching its POSITION. For example, (fetch (SCREENPOSITION POSITION) of SP12).

Screenregions

SCREENREGION [Record]

Somewhat like a region, a screenregion denotes a rectangular area in a coordinate system. Screenregions have been defined according to the following record declaration:

(RECORD SCREENREGION (SCREEN . REGION)

(SUBRECORD REGION))

Screenregions are characterized by the coordinates of their bottom left corner and their width and height. A SCREENREGION is a record with fields LEFT, BOTTOM, WIDTH, HEIGHT, and SCREEN. It can be manipulated with the standard record package facilities. There are access functions for the REGION record that return the TOP and RIGHT of the region. The user can extract the region of a screenregion by fetching its REGION. For example, (fetch (SCREENREGION REGION) of SR8).

Screenposition and Screenregion Prompting

The following functions can be used by programs to allow the user to interactively specify screenpositions or screenregions on a display screen.

(GETSCREENPOSITION *WINDOW CURSOR***)** [Function]

Similar to GETPOSITION. Returns a SCREENPOSITION that is specified by the user. GETSCREENPOSITION waits for the user to press and release the left button of the mouse and returns the cursor screenposition at the time of release. If *WINDOW* is a WINDOW, the screenposition will be on the same screen as *WINDOW* and in the coordinate system of *WINDOW*'s display stream. If *WINDOW* is NIL, the position will be in screen coordinates.

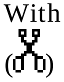
(GETBOXSCREENPOSITION *BOXWIDTH BOXHEIGHT ORGX ORGY WINDOW PROMPTMSG***)** [Function]

Similar to GETBOXPOSITION. Returns a SCREENPOSITION that is specified by the user. Allows the user to position a "ghost" region of size *BOXWIDTH* by *BOXHEIGHT* on a screen, and returns the SCREENPOSITION of the lower left corner of the screenregion chosen. A ghost region is locked to the cursor so that if the cursor is moved, the ghost region moves with it. The user can change to another corner by holding down the right button. With the right button down, the cursor can be moved across a screen or to other screens without effect on the ghost region frame. When the right button is released, the mouse will snap to the nearest corner, which will then become locked to the cursor. (The held corner can be changed after the left or middle button is down by holding both the original button and the right button down while the cursor is moved to the desired new corner, then letting up just the right button.) When the left or middle button is pressed and released, the lower left corner of the screenregion chosen at the time of release is returned. If *WINDOW* is a WINDOW, the screenposition will be on the same screen as *WINDOW* and in the coordinate system of *WINDOW*'s display stream. If *WINDOW* is NIL, the position will be in screen coordinates.its lower left corner in screen coordinates.

(GETSCREENREGION *MINWIDTH MINHEIGHT OLDREGION NEWREGIONFN NEWREGIONFNARG INITCORNERS***)** [Function]

Similar to GETREGION. Returns a SCREENREGION that is specified by the user. Lets the user specify a new screenregion and returns that screenregion. GETSCREENREGION prompts for a screenregion by displaying a four-pronged box next to the cursor arrow at one corner of a "ghost" region: . If the user presses the left button, the corner of a "ghost" screenregion opposite the cursor is locked where it is. Once one corner has been fixed, the ghost screenregion expands as the cursor moves.

To specify a screenregion: (1) Move the ghost box so that the corner opposite the cursor is at one corner of the intended screenregion. (2) Press the left button. (3) Move the cursor to the screenposition of the opposite corner of the intended screenregion while holding down the left button. (4) Release the left button.

Before one corner has been fixed, one can switch the cursor to another corner of the ghost screenregion by holding down the right button. With the right button down, the cursor changes to a "forceps" () and the cursor can be moved across a screen or to other screens without effect on the ghost screenregion frame. When the right button is released, the cursor will snap to the nearest corner of the ghost screenregion.

After one corner has been fixed, one can still switch to another corner. To change to another corner, continue to hold down the left button and hold down the right button also. With both buttons down, the cursor can be moved across a screen or to other screens without effect on the ghost screenregion frame. When the right button is released, the cursor will snap to the nearest corner, which will become the moving corner. In this way, the screenregion may be moved all over a screen and to other screens, before its size and screenposition is finalized.

The size of the initial ghost screenregion is controlled by the *MINWIDTH*, *MINHEIGHT*, *OLDREGION*, and *INITCORNERS* arguments.

(GETBOXSCREENREGION *WIDTH HEIGHT ORGX ORGY WINDOW PROMPTMSG***)** [Function]

Similar to GETBOXREGION. Returns a SCREENREGION that is specified by the user. Performs the same prompting as GETBOXSCREENPOSITION and returns the SCREENREGION specified by the user instead of the SCREENPOSITION of its lower left corner.

Color Windows and Menus

The Xerox Lisp window system provides both interactive and programmatic constructs for creating, moving, reshaping, overlapping, and destroying windows in such a way that a program can use a window in a relatively transparent fashion (see page X.XX). Menus are a special type of window provided by the window system, used for displaying a set of items to the user, and having the user select one using the mouse and cursor. The menu facility also allows users to create and use menus in interactive programs (see page X.XX). As of the LUTE release of Xerox Lisp, it is possible for the user to create and use windows and menus on the color display.

(CREATEW *REGION TITLE BORDERSIZE NOOPENFLG***)** [Function]

Creates a new window. *REGION* indicates where and how large the window should be by specifying the exterior screenregion of the window. In a user environment with multiple screens, such as a black and white screen and color screen both connected to the same machine, there is a new special problem in indicating which screen the *REGION* is supposed to be a region of. This problem is solved by allowing CREATEW to take screenregion arguments as *REGION*. For example,

```
(SETQ FOO (CREATEW (CREATE SCREENREGION
                                        SCREEN _
(COLORSCREEN)
                          LEFT _ 20
                          BOTTOM _ 210
                          WIDTH _ 290
                          HEIGHT _ 170)
                  "FOO WINDOW"))
```

creates a window titled "FOO WINDOW" on the color screen. To create a window on the black and white screen, the user should use SCREEN _ (MAINSCREEN) in the CREATE SCREENREGION expression. Note that it is still perfectly legal to pass in a *REGION* that is a region, not a screenregion, to CREATEW, but it is preferable to be passing screenregions rather than regions to CREATEW. This is because when *REGION* is a region, *REGION* is disambiguated in a somewhat arbitrary manner that may not always turn out to be what the user was hoping for.

When *REGION* is a region, *REGION* is disambiguated by coercing *REGION* to be a screenregion on the screen which currently contains the cursor. This is so that software calling CREATEW with regions instead of screenregions tends to do the right thing in a user environment with multiple screens.

(WINDOWPROP *WINDOW PROP NEWVALUE***)** [NoSpread Function]

If *PROP*='SCREEN, then WINDOWPROP returns the screen *WINDOW* is on. If *NEWVALUE* is given, (even if given as NIL), with *PROP*='SCREEN, then WINDOWPROP will generate an error. Any other *PROP* name is handled in the usual way.

(OPENWINDOWS *SCREEN***)** [Function]

Returns a list of all open windows on *SCREEN* if *SCREEN* is a screen datatype such as (MAINSCREEN) or (COLORSCREEN). If *SCREEN*=NIL, then *SCREEN* will default to the screen containing the cursor. If *SCREEN*=T, then a list of all open windows on all screens is returned.

Color Fonts

The user can create color fonts and specify in the font profile that certain color fonts be used when printing in color.

Color Font Creation

The user can create and manipulate color fonts through the same functions that are used to create and manipulate black and white fonts. This is made possible in some cases by there being new ways to call familiar font functions.

(FONTCREATE *FAMILY SIZE FACE ROTATION DEVICE NOERRORFLG CHARSET***)** [Function]

In addition to creating black and white fonts, FONTCREATE can be used to create color fonts. For example,

```
(FONTCREATE 'GACHA 10
            '(BOLD REGULAR REGULAR YELLOW BLUE)
            0 '8DISPLAY)
```

will create an 8 bit per pixel font with blue letters on a yellow background. The user indicates the color and bits per pixel of the font by the FACE and DEVICE arguments passed to FONTCREATE. DEVICE='8DISPLAY means to create an 8bpp font and DEVICE='4DISPLAY means to create a 4bpp font. A color font face is a 5 tuple,

(WEIGHT SLOPE EXPANSION BACKCOLOR FORECOLOR)

whereas a black and white font face is just a 3 tuple,

(WEIGHT SLOPE EXPANSION)

The FORECOLOR is the color of the characters of the font and the BACKCOLOR is the color of the background behind the characters that gets printed along with the characters. Both BACKCOLOR and FORECOLOR are allowed to a color name, color number, or any other legal color representation. A color font face can also be represented as a LITATOM. A three character atom such as MRR or any of the special atoms STANDARD, ITALIC, BOLD, BOLDITALIC can optionally be continued by hyphenating on BACKCOLOR and FORECOLOR suffixes. For example,

MRR-YELLOW-BLUE

BOLD-YELLOW-RED

ITALIC-90-200

BRR-100-53

are acceptable color font faces. Hence,

(FONTCREATE 'GACHA 10 'BOLD-YELLOW-BLUE 0 '8DISPLAY)

will create a color font. LITATOM FACE arguments fall into one of the following patterns:

wse	wse-backcolor-forecolor
STANDARD	STANDARD-backcolor-forecolor
ITALIC	ITALIC-backcolor-forecolor
BOLD	BOLD-backcolor-forecolor
BOLDITALIC	BOLDITALIC-backcolor-forecolor

where w=B, M, or L; s=I or R; e=R, C, or E; backcolor=a color name or color number; and forecolor=a color name or color number.

(FONTPROP *FONT PROP***)** [Function]

Returns the value of the *PROP* property of font *FONT*. Besides black and white font properties, the following font properties are recognized:

FORECOLOR The color of the characters of the font, represented as a color number. This is the color in which the characters of the font will print.

BACKCOLOR The color of the background of the characters of the font, represented as a color number. This is the color in which the the background of characters of the font will print. A font with red characters on a yellow background would have a red FORECOLOR and a yellow BACKCOLOR.

Color Font Profiles

Font profiles are the facility PRETTYPRINT uses to print different elements (user functions, system functions, clisp words, comments, etc.) in different fonts to emphasize (or deemphasize) their importance, and in general to provide for a more pleasing appearance. The user can specify that different colors of fonts be used for different kinds of elements when printing in color. A well chosen font profile will allows user to DEDIT functions, PP functions, and SEE source files in color, for example. A **FONTPROFILE** such as

```
((DEFAULTFONT 1 (GACHA 10)
        (GACHA 8)
        (TERMINAL 8)
        (4DISPLAY (GACHA 10 MRR-WHITE-RED))
        (8DISPLAY (GACHA 10 MRR-WHITE-RED)))
 (BOLDFONT 2 (HELVETICA 10 BRR)
       (HELVETICA 8 BRR)
       (MODERN 8 BRR)
       (4DISPLAY (HELVETICA 10 BRR-WHITE-MAGENTA))
```

```
        (8DISPLAY (HELVETICA 10 BRR-WHITE-MAGENTA)))
    (LITTLEFONT 3 (HELVETICA 8)
         (HELVETICA 6 MIR)
         (MODERN 8 MIR)
         (4DISPLAY (HELVETICA 8 MRR-WHITE-GREEN))
         (8DISPLAY (HELVETICA 8 MRR-WHITE-GREEN)))
    (BIGFONT 4 (HELVETICA 12 BRR)
        (HELVETICA 10 BRR)
        (MODERN 10 BRR)
        (4DISPLAY (HELVETICA 12 BRR-WHITE-BLUE))
        (8DISPLAY (HELVETICA 12 BRR-WHITE-BLUE)))
    (USERFONT BOLDFONT)
    (COMMENTFONT LITTLEFONT)
    (LAMBDAFONT BIGFONT)
    (SYSTEMFONT)
    (CLISPFONT BOLDFONT)
    ...)
```

would have comments print in green and clisp words print in blue while ordinairy atoms would print in red.

Not all combinations of fonts will be aesthetically pleasing and the user may have to experiment to find a compatible set.

The user should indicate what font is to be used for each font class by calling the function FONTPROFILE:

(FONTPROFILE *PROFILE***)** [Function]

Sets up the font classes as determined by *PROFILE*, a list of elements which defines the correspondence between font classes and specific fonts. Each element of *PROFILE* is a list of the form:

(*FONTCLASS FONT# DISPLAYFONT PRESSFONT INTERPRESSFONT* **(***OTHERDEVICE1 OTHERFONT1***)** ... **(***OTHERDEVICEn OTHERFONTn***))**

FONTCLASS is the font class name and *FONT#* is the font number for that class. *DISPLAYFONT*, *PRESSFONT*, and *INTERPRESSFONT* are font specifications (of the form accepted by FONTCREATE) for the fonts to use when printing to the black and white display and to Press and Interpress printers respectively. The appearance of color fonts can be affected by including an **(***OTHERDEVICEi OTHERFONTi***)** entry where *OTHERDEVICEi* is either 4DISPLAY or 8DISPLAY for a 4 bits per pixel or 8 bits per pixel color font and *OTHERFONTi* is a color font specification such as (GACHA 10 MRR-WHITE-RED).

FONTPROFILE [Variable]

This is the *variable* used to store the current font profile, in the form accepted by the *function* FONTPROFILE. Note that simply editing this value will not change the fonts used for the various font classes; it is necessary to execute (FONTPROFILE FONTPROFILE) to install the value of this variable.

Using Color

The current color implementation allows display streams to operate on color bitmaps. The two functions DSPCOLOR and DSPBACKCOLOR set the color in which a stream draws when the user defaults a color argument to a drawing function.

(DSPCOLOR *COLOR STREAM***)** [Function]

sets the foreground color of a stream. It returns the previous foreground color. If *COLOR* is NIL, it returns the current foreground color without changing anything. The default foreground color is MINIMUMCOLOR=0, which is white in the default color maps.

(DSPBACKCOLOR *COLOR STREAM***)** [Function]

sets the background color of a stream. It returns the previous background color. If *COLOR* is NIL, it returns the current background color without changing anything. The default background color is (MAXIMUMCOLOR BITSPERPIXEL)=15 or 255, which is black in the default color maps.

The BITBLT, line-drawing routines, and curve-drawing routines routines know how to operate on a color-capable stream. Following are some notes about them.

BITBLTing in Color

If BITBLTing from a color bitmap onto another color bitmap of the same bpp, the operations PAINT, INVERT, and ERASE are done on a bit level, not on a pixel level. Thus painting color 3 onto color 10 results in color 11.

If BITBLTing from a black-and-white bitmap onto a color bitmap, the one bits appear in the DSPCOLOR, and the zero bits in DSPBACKCOLOR. BLTing from black-and-white to color is fairly expensive; if the same bitmap is going to be put up several times in the same color, it is faster to create a color copy and then BLT the color copy.

If the source type is TEXTURE and the destination bitmap is a color bitmap, the *Texture* argument is taken to be a color. Thus to fill an area with the color BLUE assuming COLORSTR is a stream whose destination is the color screen, use (BITBLT NIL NIL NIL COLORSTR 50 75 100 200 'TEXTURE 'REPLACE 'BLUE).

Drawing Curves and Lines in Color

For the functions DRAWCIRCLE, DRAWELLIPSE, and DRAWCURVE, the notion of a brush has been extended to include a color. A BRUSH is now (BRUSHSHAPE BRUSHSIZE BRUSHCOLOR). Also, a brush can be a bitmap (which can be a color bitmap).

Line-drawing routines take a color argument which is the color the line is to appear in if the destination of the display stream is a color bitmap.

(DRAWLINE *X1 Y1 X2 Y2 WIDTH OPERATION STREAM COLOR***)** [Function]

(DRAWTO *X Y WIDTH OPERATION STREAM COLOR***)** [Function]

(RELDRAWTO *X Y WIDTH OPERATION STREAM COLOR***)** [Function]

(DRAWBETWEEN *POS1 POS2 WIDTH OPERATION STREAM COLOR***)** [Function]

If the *COLOR* argument is NIL, the DSPCOLOR of the stream is used.

Printing in Color

Printing only works in REPLACE mode. The characters have a background color and a foreground color determined by the font face of the font the characters are being printed with.

Example of printing to an 8bpp color screen:

```
(SETQ FOO (CREATEW (CREATE SCREENREGION
                                        SCREEN _
(COLORSCREEN)
                              LEFT _ 20
                              BOTTOM _ 210
                              WIDTH _ 290
                              HEIGHT _ 170)
                    "FOO WINDOW"))
(DSPFONT (FONTCREATE 'GACHA
                     10
                     'MRR-YELLOW-GREEN
                     0
                     '8DISPLAY)
     FOO)
(PRINT 'HELLO FOO)          ; will print in green against a yellow
background.
```

Operating the Cursor on the Color Screen

The cursor can be moved to the color screen. The cursor can be moved to the color screen by sliding the cursor off the left or right edge of the black and white screen on to the color screen or by calling function CURSORPOSITION or CURSORSCREEN.

(CURSORPOSITION *NEWPOSITION* **- -)** [Function]

NEWPOSITION can be a position or a screenposition.

(CURSORSCREEN *SCREEN XCOORD YCOORD***)** [Function]

Moves the cursor to the screenposition determined by *SCREEN*, *XCOORD*, and *YCOORD*. *SCREEN* should be the value of either (COLORSCREEN) or (MAINSCREEN).

While on the color screen, the cursor is placed by doing BITBLTs in software rather than with microcode and hardware as with the black and white cursor. It is automatically taken down whenever an operation is performed that changes any bits on the color screen. The speed of the color cursor compares well with that of the black and white cursor but there can be a noticeable flicker when there is much input/output to the color screen. While the cursor is on the color screen, the black-and-white cursor is cleared giving the appearance that there is never more than one cursor at a given time.

Miscellaneous Color Functions

(COLORIZEBITMAP *BITMAP 0COLOR 1COLOR BITSPERPIXEL***)** [Function]

creates a color bitmap from a black and white bitmap. The returned bitmap has color number *1COLOR* in those pixels of *BITMAP* that were one and *0COLOR* in those pixels of *BITMAP* that were zero. This provides a way of producing a color bitmap from a black and white bitmap.

(UNCOLORIZEBITMAP *BITMAP COLORMAP***)** [Function]

creates a black and white bitmap from a color bitmap.

(SHOWCOLORTESTPATTERN *BARSIZE***)** [Function]

displays a pattern of colors on the color display. This is useful when editing a color map. The pattern has squares of the 16 possible colors laid out in two rows at the top of the screen. Colors 0 through 7 are in the top row, and colors 8 through 15 are in the next row. The bottom part of the screen is filled with bars of *BARSIZE* width with consecutive color numbers. The pattern is designed so that every color has a border with every other color (unless *BARSIZE* is too large to allow room for every color—about 20).

Color/Advanced Graphics Features Brainstorming

In the list below, rank (between 1 and 5, 1 highest to 5 lowest) the priority of each item or feature you would be interested in. OR feel free to add to the list of things you might like to see.

- Color windows
- Color fonts
- Move windows from b/w to color display?
- Multi-monitor Suns?
- Sun color hardware
    - CG2
    - CG3
    - CG4
    - CG5
    - Special graphics accelerators
- Need separate fonts for fore, backgrounds colors (versus font or stream attribute)
- Color bitmaps will eat up 32M space quickly (windows, fonts)
    - Remote bitmaps (outside of 32M address space - can't save in sysout)
- New Opcodes
    - 2&3 D geometric transforms (avoids floating pt. boxing)
    - BitMapBit
    - TEdit Color
    - Your suggestion here
- Cleanup texture/shade/color controversies
- Application support
    - raytracers
    - renderers
- Application type
    - User Interface
    - 2 D
    - 3 D
    - Animation
    - Image analysis
- Other I/O devices
    - film recorders
    - color hardcopy
        - InterPress
        - Postscript
        - Others?
    - scanners
    - digitizing tablet
- File I/O of various formats
    - AIS
    - Others?
- Better quality graphics capabilities
    - contour fonts
    - better polygon capabilities (e.g. "woodgrain texture fill")
- Speed requirements? (often need to process *lots* of pixels)
- Can logout, restart from color screen? Are there multiple resolutions (multiple BPP) of color?
- Integration with other modules
    - Rooms

- ScreenPaper
- ACE (animation editor)
- Notecards
- LOOPS
- TEdit
- Sketch

Compatibility with Medley1.1 sysouts, data structures
Kickstarting (integrated w. current scripts (ldeether -c calls ldecolor?)
Light weight high quality bitmap editor
Resolution independent streams
Postscript interpreter
Network ray tracer
Immediate bare-bones capabilities
(versus) Not so immediate richer capabilities
Flood fill

LAFITEABBREV

By: Lennart Lövstrand
(Lovstrand.EuroPARC@Xerox.COM)

Requires: LAFITE, NSMAIL

INTERNAL

This document last edited on December 21, 1988.

INTRODUCTION

As the name suggests, LAFITEABBREV gives you the ability to refer to mail addresses using personal, easy to remember abbreviations. If you define (say) "Joe" to stand for "John Doe:Xyzzy North:ACME Corporation", then all you need to type to send a message to Joe is *Joe*. If you don't want to be bothered with his long address even when you read messages from him, you can optionally let the abbreviation apply both ways. When receiving messages from him (or where you both are recipients), his full address will be replaced by your abbreviation and thus only show as simple *Joe* in all header lines and the Lafite browsers.

In addition to this rather straight forward string-to-string text substitution, LAFITEABBREV will also do pattern matching using multiple wildcards in both the abbreviation and fully expanded strings. This is of course particularly useful for cases when many people share the same substring in their addresses, such as the NS domain. Say for example that Jane Doe worked at the same place as her brother. Then we could abbreviate "*:Xyzzy North:ACME Corporation" as "*:X:ACME" and refer to John as "John Doe:X:ACME" and Jane as "Jane Doe:X:ACME". In fact, we could even create the abbreviation "* Doe" ⇒ "* Doe:Xyzzy North:ACME Corporation" and refer to our friends as just "John Doe" and "Jane Doe" with the obvious expansions.

Abbreviations are purely personal; all other recipients of the message will see the real, expanded addresses. The abbreviations are on a strictly one-to-one mapping basis, ie. you can't implement private DLs by making an abbreviation expand into more than one address. As is the case with normal mail addresses, all abbreviations and expansions are case-insensitive.

EXAMPLES

An individual abbreviation:	Sir	⇒	Tom Moran:EuroPARC:RX
A group abbreviation:	* (APU)	⇒	*@MRC-APU.CAM.AC.UK:GV:Xerox
Hiding the local domain:	*	⇐	*:EuroPARC:RX
Hiding gateway syntax:	*@*.EDU	⇔	*%*:EDU:Xerox

MODULE EXPLANATION

LAFITEABBREV advises the Lafite NSMAIL routines that encode and decode string addresses to NSNAMEs. It operates transparently and automatically with respect to the rest of the mail system while being controlled by the variables mentioned below. The most central of these is the list of abbreviations itself, LAFITE.ABBREVS:

LAFITE.ABBREVS [Variable]

This variable contains all abbreviations and their corresponding real addresses. It is formed as a list of translations, where each translation is a list of (up to) four elements: the abbreviation, the full address, an optional direction, and an optional comment. Both the abbreviation and full address should be strings; the direction may be either of `:IN`, `:OUT`, `:BOTH`, `:NONE`, or left empty in which case it defaults to `:BOTH`. Finally, the comment, if present, should be in normal Interlisp format, ie. `(*` *text*`)`. It has no functional purpose and serves only as a memory aid. A full translation thus looks like:

(*abbrevation full-address direction comment*)

eg: `("Pete" "Piotr Kropotkin:RusskiPARK:ZeRoks" :OUT (* ; "Old chap"))`

which means that an address of *Pete* will be expanded to *Piotr Kropotkin:RusskiPARK:ZeRoks* whenever you send a message to him, but his full address will always be presented as such when receiving mail from him or whenever his address is mentioned in the header lines of a message you receive. For an explanation on translation directions, read more below.

The default list of translations is:

```
*@*     "*%*":GV:Xerox   :IN
*@*      *%*:GV:Xerox
*@*.*     *%*:*:Xerox    :IN
*.pa      *:PA:Xerox
```

This will present most ARPA Internet addresses in their proper form and also allow you to see Palo Alto Grapevine addresses as if you were on Grapevine. Note that the first rule above is necessary because some external GV addresses are being enclosed in double quotes by the GV/XNS Mail Gateway (for no appearant reason, one might add, since there exist no common conventions for parsing such constructs and it technically is malforming the address).

LAFITE.ABBREV.DIRECTIONS [Variable]

This variable controls the overall behaviour of LAFITEABBREV. It can take either of the following values:

`:IN`	Translate full addresses to abbreviations when receiving messages.
`:OUT`	Translate abbreviations to full addresses when sending messages.
`:BOTH`	Both of the above.
`:NONE`	Neither of the above, ie. don't ever do any translations at all.

The setting of LAFITE.ABBREV.DIRECTIONS limits the scope of individual translation rules, so if you for example set LAFITE.ABBREV.DIRECTIONS to `:OUT`, no translations will ever be done on incoming messages, not even if a rule has an explict direction of `:IN` or `:BOTH`.

The default value of LAFITE.ABBREV.DIRECTIONS is `:BOTH`.

LAFITE.ABBREV.MOVE.GAZE.RIGHT [Variable]

A quite handy application of LAFITEABBREV is to present ARPA Internet (RFC-822) addresses in their proper form, ie. translating addresses like Jakobsson%Score.Stanford:EDU:Xerox to Jakobsson@Score.Stanford.EDU and vice versa. Unfortunately, since the pattern matching algorithm uses a strict left-to-right order, an address like user%host%foo.bar:EDU:Xerox would end up translated as user@host%foo.bar.EDU, which is not quite what was inteded. However, if LAFITE.ABBREV.MOVE.GAZE.RIGHT is set to a non-NIL value, translations resulting in a percent sign coming anywhere after an atsign will be automatically changed so that the rightmost percent sign is substituted for an atsign instead. Don't worry too much if you didn't understand any of the above; just think of it as applied magic (aka a *kludge*).

The default value of LAFITE.ABBREV.MOVE.GAZE.RIGHT is `T`.

LAFITE.ABBREV.TRACE [Variable]

Set this variable to T or a window of your choice (eg the PROMPTWINDOW) to trace all translations that are done for you. Reset to NIL to stop tracing.

The default value of LAFITE.ABBREV.TRACE is `NIL`.

(LAFITE.ABBREV *ADDRESS DIRECTION*) [Function]

This is the function that does the actual translations; it is used automatically by LAFITEABBREV's advices, but mentioned here if you want to test it manually or use it elsewhere. The address should be the string to expand/abbreviate and the direction either of `:IN` or `:OUT`, indicating whether this address is about to be sent or received. The result is the translated address.

(LAFITE.ABBREV.MATCH *PATTERN STRING TEMPLATE*) [Function]

This is a small, case insensitive pattern matching algorithm supporting multiple wildcards on both the pattern and the template side. Asterisks will match substrings of zero or more characters, anything else has to match literally for the comparison to succeed. The result is constructed using the template, if supplied. Some examples:

PATTERN	**STRING**	**TEMPLATE**	**RESULT**
"Foo"	"FOO"	NIL	"FOO"
"Foo"	"FOO"	"Bar"	"Bar"
"Foo*Bar"	"FOOBAZBAR"	NIL	"FOOBAZBAR"
"Foo*Bar"	"FOOBAZBAR"	"1*2"	"1BAZ2"
"Foo*B*ar"	"FOOBAZBAR"	"1*2*3"	"12AZB3"
"Foo*B*ar"	"FOOBAZBA"	"1*2*3"	NIL

BUGS

Only XNS is supported (ie. GV addresses aren't touched). Messages from you won't be presented as "To: <Recipients>" in the browser if you abbreviate your own name. No nice'n'cuddly user interface included.

LAFITE-INDENT

By: smL (Lanning.pa@Xerox.com)

INTERNAL

This document last edited December 3, 1987.

INTRODUCTION

It is common practice in conversations conducted via electronic mail to include bits and pieces of previous messages in reply messages. The included text is indented to set it apart visually from the main body of the message. LAFITE-INDENT provides some support for this style of e–mail use.

USING LAFITE-INDENT

LAFITE-INDENT adds the item "Indent" to the default TEdit menu. Selecting the "Indent" item indents the current selection. In addition, it inserts carriage–returns in the text to ensure that no line in the selection is too long. The particular method of indentation is controlled by global variables described below. The "Indent" menu item also contains a number of subitems described below.

Indent [TEdit menu item]

This item will indent the current selection by prepending (the value of) *TEDIT-INDENT-STRING* to each line. To ensure that resulting text is properly formatted, lines will be broken to contain no more than *TEDIT-INDENT-LINE-LENGTH* characters. Existing carriage returns in the current selection may be replaced by spaces. For example the result of indenting the highlighted text below:

```
Date: 9 Sep 86 13:34:12
Subject: Eat?
To: JFrench
From: Wu

Hi Jane!
I was wondering if you might be able to make it over to our house tonight for dinner.
Carol's got a turkey that she wants to BBQ, Hank is coming over with a couple of bottle
of his home brew beer, and Diane's got a salad fresh from her garden.

Even if you can't make it for dinner, you might want to stop by to visit.  Remember
Crazy Tom from Berkeley?  He's in the area on a job interview and he'll be there.

Let me know soon...
```

```
Date: 9 Sep 86 13:34:12
Subject: Eat?
To: JFrench
From: Wu

    Hi Jane! I was wondering if you might be able to make it over
    to our house tonight for dinner.  Carol's got a turkey that she
    wants to BBQ, Hank is coming over with a couple of bottle of his
    home brew beer, and Diane's got a salad fresh from her garden.
```

```
        Even if you can't make it for dinner, you might want to stop by
        to visit.  Remember Crazy Tom from Berkeley?  He's in the area on a
        job interview and he'll be there.

    Let me know soon...
```

Example 1: Before and after "Indent"

Indent & keep lines [TEdit menu item]

Like Indent, Indent & keep lines will indent the current selection. Unlike Indent, returns in the current selection are kept. This can be useful when indenting the headers of mail messages. For example the result of "Indent & keep lines" on the following text:

```
    Date: 9 Sep 86 13:34:12
    Subject: Eat?
    To: JFrench
    From: Wu

    Hi Jane!
```

```
        Date: 9 Sep 86 13:34:12
        Subject: Eat?
        To: JFrench
        From: Wu

    Hi Jane!
```

Example 2: Before and after "Indent & keep lines"

If Indent had been used instead of "Indent & keep lines", the lines would have been run together, resulting in:

```
        Date: 9 Sep 86 13:34:12 Subject: Eat? To: JFrench From: Wu
```

Set indent [TEdit menu item]

The "Set indent" message lets you change the current indentation string. It will prompt you for the new value of *TEDIT-INDENT-STRING*.

Unindent [TEdit menu item]

Unindent will remove all indentation from the current selection. In the process, it will replace all "extra" carriage returns by spaces. For example:

```
    Date: 9 Sep 86 13:34:12
    Subject: Eat?
    To: JFrench
    From: Wu

        Hi Jane! I was wondering if you might be able to make it over
        to our house tonight for dinner.  Carol's got a turkey that she
        wants to BBQ, Hank is coming over with a couple of bottle of his
        home brew beer, and Diane's got a salad fresh from her garden.

        Even if you can't make it for dinner, you might want to stop by
        to visit.  Remember Crazy Tom from Berkeley?  He's in the area on a
        job interview and he'll be there.

    Let me know soon...
```

```
Date: 9 Sep 86 13:34:12
Subject: Eat?
To: JFrench
From: Wu

Hi Jane! I was wondering if you might be able to make it over to our house tonight for
dinner.  Carol's got a turkey that she wants to BBQ, Hank is coming over with a couple
of bottle of his home brew beer, and Diane's got a salad fresh from her garden.

Even if you can't make it for dinner, you might want to stop by to visit.  Remember
Crazy Tom from Berkeley?  He's in the area on a job interview and he'll be there.

Let me know soon...
```

Example 3: Before and after "Unindent"

Open line [TEdit menu item]

"Open line" will open a blank line at the current position in the textstream. Text following the current position maintain its distance from the left margin. For example:

```
Date: 9 Sep 86 13:51:20
Subject: Re: Eat?
To: Wu
From: JFrench

    Hi Jane! I was wondering if you might be able to make it over
    to our house tonight for dinner.  Carol's got a turkey that she
    wants to BBQ, Hank is coming over with a couple of bottle of his
    home brew beer, and Diane's got a salad fresh from her garden.
```

```
Date: 9 Sep 86 13:51:20
Subject: Re: Eat?
To: Wu
From: JFrench

    Hi Jane! I was wondering if you might be able to make it over
    to our house tonight for dinner.  Carol's got a turkey that she
    wants to BBQ, Hank is coming over with a couple of bottle
                                                          of his
    home brew beer, and Diane's got a salad fresh from her garden.
```

Example 4: Before and after "Open line"

Insert <RETURN>s [TEdit menu item]

This will replace TEdit's line breaks with real carriage returns (in the current selection). This way you know what line breaks recipients of your message will see.

Break long lines [TEdit menu item]

This item inserts <RETURN>s in the text, making sure that no line contains more than *TEDIT-INDENT-LINE-LENGTH* characters. Functionally, this is just like the "Indent" item, but without the indentation.

VARIABLES

The following variables can be changed to tailor how text is indented. They are defined as INITVARS in LAFITE-INDENT, so they can be given different values before or after loading the LAFITE-INDENT.

TEDIT-INDENT-STRING [GlobalVar]

This is the string that is used to indent each line of text. The default value is (the value of) `(ALLOCSTRING 4 " ");` a string consisting of four spaces.

TEDIT-INDENT-LINE-LENGTH [GlobalVar]

This is the maximum number of characters on a line that the indentation will allow. Extra line breaks will be added to ensure that no line has more than *TEDIT-INDENT-LINE-LENGTH* characters. The default value is `72`.

BUGS

Unfortunately, these operations are not (always) UNDOable.

LAFITETIMEDDELETE

By: John Maxwell (Maxwell.pa)

REQUIRES: LAFITE

INTERNAL

This document last edited on October 7, 1987.

INTRODUCTION

The package LafiteTimedDelete allows users to specify expiration dates on their mail messages as they read them. After a message has expired, it will be marked for deletion the next time the user invokes the command "Delete Expired Msgs". The package is useful for specifying deletion dates on dated information such as announcements of talks. It can also be used as a "sunset" clause on certain messages, saying in effect that if a message hasn't been acted on by this date, mark it for deletion. Since the command for deleting expired messages only marks them for deletion and doesn't actually delete them, the user can always intervene and save the message from deletion.

SETTING EXPIRATION DATES

To set an expiration date, simply select the message(s) that you wish to have deleted at some future time, and then click the "Delete" command on the Lafite browser with the middle button. This will produce a menu of durations like this:

Selecting a duration will determine how long until the message(s) expire, measured from the current day. If no selection is made, then the operation is aborted.

When a message is marked for future deletion, a number from 1 to 9 will appear in the mark field to the left of the message id. This number indicates how long until the message should expire, *measured from the date of the message*. The number is approximately the logarithm of the number of days until the message expires. Thus "1" means 1 day, "2" means 2 days, "3" means 4 days, "4" means 1 week, "5" means 2 weeks, "6" means 1 month, "7" means 2 months, "8" means 4 months, and "9" means 8 months.

Selecting the duration "now" is equivalent to normal deletion. Selecting the duration "forever" is equivalent to undeletion, with the side effect that all expiration marks are removed.

DELETING EXPIRED MESSAGES

To delete all of the messages that have expired, invoke the "Delete Expired Msgs" command in the browser's middle button menu. (The middle button menu is obtained by holding down the middle button while over the black bar in the mail folder.) The program will then examine all of the messages in that folder, looking for messages that have expired. When it finds a message that has expired, it will mark it to be deleted. Finally, it will print in the browser's prompt window the number of messages that it marked for deletion.

Messages expire at 12 noon on the Nth day from the date given in the date field of browser. Thus if on Wednesday, October 7th you mark a message sent that day to be deleted in two days, then the message would be deletable after 12 noon on Friday, October 9th. However, the messages don't actually get marked for deletion until you invoke the command "Delete Expired Msgs". You usually only need to invoke this command once per day since no new messages will expire later on, so if you are unhappy with 12 noon as an expiration time, you can move it by only invoking the command after the day is over (or early in the morning the next day.)

CAVEATS

Since LAFITETIMEDDELETE uses the mark field of the message header to save the information about how long until the message expires, you may run into conflicts with other uses of the mark field. For instance, if you set the expiration date on a message and then forward it, you will lose the expiration information because the forwarder puts an "f" in the mark field. If you notice this, you can restore the mark manually by clicking the mark field and typing a new number.

Also, since LAFITETIMEDDELETE is limited in the number of distinctions that it can make in the time until expiration, it may set the expiration of a message to a time much later than you might want. For instance, suppose that you are reading a message that was mailed yesterday and you want to set its expiration date to one week from today. Since it can only record expirations from the date of the message and not from the current date, it must set the message to expire in eight days. If the program used the one week expiration mark, then the message would expire in seven days, or six days from today. Since that might expire the message before you intend, the program plays it safe and uses the two week expiration mark. A six day expiration might be OK with the user, but it would be awkward to ask him, especially if there were multiple messages selected. If the user is dissatisfied with the mark, he can always change it manually as described above.

Please send all comments, questions, and bug reports to Maxwell.pa.

MAILSCAVENGE

For Scavenging Mail Folders

The Lisp Library package `MAILSCAVENGE` is used to rebuild the internal pointers in a mail file that has been damaged. Lafite generally reports ‘‘Can’t parse file’’ and terminates its Browse command when it detects damage in a file. The simplest remedy is to call `MAILSCAVENGE`, then browse the file again.

(`MAILSCAVENGE` *FILENAME ERRORMSGSTREAM TEMPDIR*) [Function]

This function scavenges the file named *FILENAME*. *FILENAME* defaults to extension `MAIL` and your Lafite directory, exactly as with Lafite’s Browse command. If *ERRORMSGSTREAM* is specified, it is a stream on which `MAILSCAVENGE` writes information about what it is doing to correct the file. *TEMPDIR* is a host/directory specification for where `MAILSCAVENGE` should write its intermediate file. *TEMPDIR* defaults to {`DSK`}, unless you are on an 1108 without a local disk file system, in which case it defaults to the same directory as *FILENAME*.

When `MAILSCAVENGE` finishes, it asks if you want to replace the original file with the scavenged file. If you reply "No", `MAILSCAVENGE` returns the name of the temporary file it wrote, which you can then rename or delete as you wish. Ordinarily, you should reply "Yes", unless you find the report of what `MAILSCAVENGE` had to correct to be suspicious.

(`MAILSCAVENGE.IN.PLACE` *FILENAME ERRORMSGSTREAM*) [Function]

This is similar to `MAILSCAVENGE`, but destructively scavenges the file in-place. This is faster than `MAILSCAVENGE`, but you have to be brave and assume `MAILSCAVENGE` is not overwriting anything valuable as it scans the file.

MIME decoding package, by Ron Kaplan, 2/2000.

This file recognizes and decodes email messages that conform to the MIME internet standard. It interfaces to Lafite so that it can parse the MIME headers and recognize various kinds of attachments. The attachments are represented in the message display window as an image-object box. Clicking on that box will let you extract the attachment and write it to a file on a directory which is obtained by evaluating the value of ATTACHMENTDIR. This is initialized to

```
(CONCAT "{dsk}/tilde/"  (L-CASE (USERNAME)) "/attachments")
```

which, for example, would evaluate to "{dsk}/tilde/kaplan/attachments".

The MIME package tries to show text attachements in-line, and it offers a Print option for attachments (or even just included text) that it recognizes as Postscript.

The extracted files can be accessed by standard PC software packages. For certain Mac-derived application types, it tries to provide the appropriate PC extension so that double clicking will launch the appropriate application.

NSMAIL

INTRODUCTION

The module NSMAIL implements the protocols to allow Lafite to be used to send and retrieve Xerox NS Mail. Load the file NSMAIL.LCOM. To run this in Lyric, you must have loaded the LispUsers module NSRANDOM as well (q.v. for important loading information). If you don't have NSRANDOM loaded, you can't use the "Put to file" command described below.

If you have both Grapevine and NSMAIL implementations loaded, you must set Lafite's mode to NS. Use the "NS Mode" subitem underneath Lafite's Quit command, or call (LAFITEMODE 'NS). You must also be a registered NS user, and have a mailbox.

ATTACHMENTS

The main difference between this and earlier versions of NSMAIL is that "attachments" are no longer left in your mailbox to be read later with, for example, Viewpoint. Instead, Lafite retrieves the entire attachment and encapsulates it into an image object that is enclosed as part of the text message, immediately following the header. A typical attachment appears in a mail message as:

Attachment: Viewpoint Document

If you click inside the object with any mouse button, you are offered a menu of things you can do with the attachment. The choices vary according to the type of attachment:

View as text — This brings up a window in which is displayed the raw content of the attachment as ascii bytes. Runs of non-ascii bytes are replaced by nulls to reduce the amount of garbage. Some attachments are utter gibberish, but some, such as Viewpoint documents and Interpress masters, contain sections that are plain text. With this command, you may be able to decide whether you care to do anything further with the attachment. (Sorry, there is no Viewpoint to TEdit converter, nor are there plans for one.)

Put to file — This prompts you for a file name, and creates a file to contain the attachment. The file must be on an NS file server for this command to be very useful; otherwise, information will be lost. Once the file is so stored, you can retrieve it from Viewpoint and manipulate it just as if you had originally retrieved it as mail in Viewpoint.

Send to Printer — This command is only available for attachments that are in the form of an Interpress master. The command prompts you for a printer (must be one that accepts Interpress, of course), and sends the attachment to it for printing.

Expand folder — This command is only available for attachments that are in the form of a "folder". A folder is a mechanism for collecting several objects into a single one. The Expand folder command splits the attachment up into its component objects, each of which can be manipulated in the same way as a top-level attachment. For example, if the folder contains an Interpress master, you can print it.

If you use the Put to file command on a folder, the name component of the file name you type will be treated as the name of a new subdirectory, and the components of the folder will appear as files in that subdirectory. For other types of attachments, Put to file (usually) produces an ordinary (non-directory) file.

Messages containing attachments are otherwise just like formatted messages—you can move them to other folders, and you can forward them (assuming the mail is received by another Lafite recipient and did not have to pass through a mail gateway).

There is currently no mechanism for creating your own attachments to send to other users.

MISCELLANY

If you prefer the old behavior of leaving the attachments behind in the mailbox, set the variable NSMAIL.LEAVE.ATTACHMENTS to T, but this use is discouraged. You must take care to regularly retrieve your mail from somewhere (such as Viewpoint) that will flush out all the mail; otherwise, the mail with attachments, whether you want them or not, accumulate on the server.

When in NS mode, Lafite will want your NS login identity. Normally, if your NS password differs from your default password, you will be prompted to login. You can also call (LOGIN '|NS::|) yourself to set your NS login.

You can freely intermix Grapevine and NS mail in the same mail folder if you like, but the Answer command always treats the selected message as if it were one in the current mode. So if you try to answer a Grapevine message while in NS mode, some confusion may result. Also, the status window always shows you mail status only of the current mode.

UNIXMAIL

By: Bob Bane (Bane.mv@envos.Xerox.com)

INTRODUCTION

UNIXMAIL is a new mail sending and receiving mode for Lafite. It sends mail via Unix hosts using the SMTP mail transfer protocol and can receive mail either by reading a Unix mail spool file or by calling the Berkeley mail program.

INSTALLATION

Turn Lafite off, load the file UNIXMAIL, make sure UNIXMAIL is configured appropriately (check the settings of the variables below, and make sure any other modules UNIXMAIL may need are loaded), then restart Lafite. If you are running Lafite on a machine that is isolated from the Xerox mail environment, you will probably want to set the variable LAFITE.USE.ALL.MODES to NIL and call (LAFITEMODE ’UNIX) before you turn Lafite back on.

CONFIGURING

See **SENDING MAIL** and **RECEIVING MAIL** below for the exact meanings of the variables you will be asked to set.

D-machines:

UNIXMAIL.SEND.MODE must be set to SOCKET and UNIXMAIL.SEND.HOST must be set to the name of a TCP host that will accept SMTP connections. UNIXMAIL.RECEIVE.MODE must be set to SPOOL and UNIXMAIL.SPOOL.FILE must be set to the pathname of your Unix mail spool file.

Unix-based emulators:

The default values of UNIXMAIL.SEND.MODE and UNIXMAIL.RECEIVE.MODE (PROCESS and SPOOL, respectively) will work if you normally send and receive mail from the machine where Medley is running.

OTHER MODULES YOU MAY NEED

UNIXMAIL may need other library modules to work. The modules needed vary depending on what hardware you are using:

D-machines:

TCP is mandatory for Unix sending and may be used for Unix receiving, NFS is optional for Unix receiving

Unix-based emulators:

one of TCPOPS or UNIXCOMM is mandatory for sending

SENDING MAIL

UNIXMAIL can send mail in one of two ways, depending on the setting of UNIXMAIL.SEND.MODE:

UNIXMAIL.SEND.MODE [Variable]

If its value is the atom PROCESS, UNIXMAIL will send mail by doing SMTP with a Unix process-stream, normally running `/usr/etc/mconnect`. This option only works on Medley running one of the Unix-based emulators.

If its value is the atom SOCKET, UNIXMAIL will send mail by doing SMTP with a TCP host. For this to work, an appropriate version of TCP must be loaded: either the TCP library module for D-machines or the TCPOPS library module for emulators that support it.

UNIXMAIL.SEND.MODE defaults to PROCESS.

Each of these send modes can be configured as well:

UNIXMAIL.SEND.PROCESS [Variable]

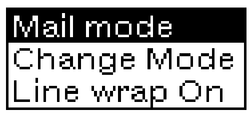
When UNIXMAIL.SEND.MODE is PROCESS, the value of this variable is the program run to create the SMTP process-stream. Initially the string `"/usr/etc/mconnect"`

UNIXMAIL.SEND.HOST [Variable]

When UNIXMAIL.SEND.MODE is SOCKET, the value of this variable is the name of the host UNIXMAIL will attempt to contact via TCP to open an SMTP stream over socket 25. Initially NIL; on a Unix-based emulator this means to try the machine Medley is running on. This variable must be set when running on a D-machine.

UNIXMAIL.WRAP.LINES [Variable]

This flag controls whether or not outgoing mail has its lines word-wrapped to a fixed length. It defaults to T, meaning word-wrapping is done. UNIXMAIL patches the Change Mode menu entry of the standard Lafite message form, adding an entry for toggling UNIXMAIL.WRAP.LINES:



UNIXMAIL.WRAP-LIMIT [Variable]

If UNIXMAIL.WRAP.LINES is true, this variable is the length in characters to which lines are wrapped. Default value is 72.

UNIXMAIL.TABWIDTH [Variable]

If UNIXMAIL.WRAP.LINES is true, this variable is the width tab characters are assumed to expand into for word-wrapping purposes. Default value is 8.

UNIXMAIL.RECIPIENT.PATTERNS [Variable]

This variable is a list of patterns that are applied to outgoing UNIXMAIL addresses; it can be used to catch bogus addresses and modify them before sending. List entries are of the form (*pattern* . *function*) where *pattern* is a list of arguments that will be passed along with the address to STRPOS; if STRPOS returns non-NIL, *function* is called with the address and the result replaces the address. For example, if mail from UUCP host `black-silicon` is arriving via path `mimsy!black-silicon`, but the address in its headers is missing the `mimsy`, this entry on UNIXMAIL.RECIPIENT.PATTERNS will add it back:

```
(("black-silicon!" NIL NIL T) LAMBDA (R) (CONCAT "mimsy!" R)
```

This means that whenever `(STRPOS "black-silicon!"` *`address`* `NIL NIL T)` returns non-NIL, *`address`* will have `"mimsy!"` prepended before the message is sent.

RECEIVING MAIL

UNIXMAIL can receive mail in one of two ways, depending on the setting of UNIXMAIL.RECEIVE.MODE:

UNIXMAIL.RECEIVE.MODE [Variable]

If its value is the atom SPOOL, UNIXMAIL will receive mail by reading a Unix mail spool file.

If its value is the atom MAILER, UNIXMAIL will receive mail by running a Berkeley mailer as a Unix process-stream, normally `/usr/ucb/mail`. This option only works on Medley running one of the

Unix-based emulators, and is a bit slower than SPOOL mode; it is primarily useful when you wish to occasionally switch between Lafite and the Berkeley mailer.

UNIXMAIL.RECEIVE.MODE defaults to SPOOL.

Each of these receive modes can be configured as well:

UNIXMAIL.RECEIVE.PROCESS [Variable]

When UNIXMAIL.RECEIVE.MODE is MAILER, the value of this variable is the program run to create the SMTP process-stream. Initially the string `"/usr/ucb/mail -N"`; the `-N` means to not print any banner or read any intialization file on starting the mailer.

UNIXMAIL.DONT.RECEIVE.STATUS [Variable]

When UNIXMAIL.RECEIVE.MODE is MAILER, the value of this variable is a set of message status letters; UNIXMAIL will leave behind any message whose status is included**.** Initially `""`, which means to read all messages regardless of status; another useful value would be `"O"` which means leave old messages behind.

UNIXMAIL.SPOOL.FILE [Variable]

When UNIXMAIL.RECEIVE.MODE is SPOOL, the value of this variable is the file UNIXMAIL will receive mail from. Any time this file has characters in it, Lafite will say you have new mail; when Lafite gets mail from this file, it will read all messages in the file and then set its size to zero. Initially NIL; on a Unix-based emulator this means to try the file `"{UNIX}/usr/spool/mail/`**`username`**`"`, where **`username`** is the value of (UNIX-USERNAME). To access a Unix mail spool file from a D-machine, it will probably be necessary to load and configure either the TCP or NFS modules and then set UNIXMAIL.SPOOL.FILE appropriately.

MAIKOCOLOR

Introduction

This module is the Envos Lisp software driver for running the COLOR module under Maiko (Maiko is the MACHINETYPE of Medley running on the Sun workstations). The machine independent functionality is provided by and documented with the COLOR module. There are no MAIKOCOLOR functions that the user needs to call directly. The user calls functions described in the COLOR documentation.

Requirements

In order to run COLOR, you need a Sun 3 or Sun 4 with a color display (CG 4), plus a color emulator (lde). COLOR will *not* run on a non-color emulator, attempting to use color capabilities will cause an error trap into URaid. You additionally need:

COLOR
LLCOLOR

Installation

MAIKOCOLOR may be loaded into a sysout running on any D-Machine (or non-color emulator), as long as color is not initialized. Thus an initial sysout can be made which runs on all systems, by loading COLOR, and then writing the sysout.

To install, simply (FILESLOAD ’MAIKOCOLOR), which will load all the additional files necessary for running color.

Initialization

To actually use color, the user must be running on a color capable emulator (lde).

To Initialize color running under Maiko, simply type

(COLORDISPLAY ’ON ’MAIKOCOLOR)

Lisp will permanently allocate a color "screen", and will attempt to map the color frame buffer to that screen.

At this point, the user should refer to the documentation for COLOR.

Known Bugs

NOTE: As this is currently unreleased software undergoing active development, this list of bugs should not be construed as being a limitation or defect of the final product. This list is only included to point out the current state of affairs of the software.

1. Color screen is never GC'ed

2. Color TEdit slower than B/W TEdit

3. Can't create big color windows

POSTSCRIPTSTREAM

By: Matt Heffron (mheffron@orion.cf.uci.edu)

INTRODUCTION

The PostScript package defines a set of imageops for printers which understand the PostScript page description language by Adobe. At Beckman we have successfully used TEdit, Sketch, LISTFILES, and HARDCOPYW to an Apple LaserWriter and an AST TurboLaser PS. The PostScript imagestream driver installs itself when it is loaded. All symbols in the PostScript driver are located in the INTERLISP: package.

VARIABLES

POSTSCRIPT.FONT.ALIST [InitVariable]

POSTSCRIPT.FONT.ALIST is an ALIST mapping Xerox Lisp font names into the root names of PostScript font files. It is also used for font family coercions. The default value should be acceptable for any of the fonts which are built into the Apple Laserwriter.

POSTSCRIPTFONTDIRECTORIES [InitVariable]

POSTSCRIPTFONTDIRECTORIES is the list of directories where the PostScript .PSCFONT font files can be found. The default value is: ("{DSK}/usr/local/lde/fonts/postscript/") on a Sun or IBM workstation and ("{DSK}<LISPFILES>FONTS>PSC>") for other cases .

POSTSCRIPT.DEFAULT.PAGEREGION [InitVariable]

POSTSCRIPT.DEFAULT.PAGEREGION indicates the area of the page to use for text file listings (i.e. LISTFILES). It is in units of 100'ths of points. The default value is: (4800 4800 52800 70800), which gives left and bottom margins of 0.75 inch and top and right margins of 0.5 inch on 8.5 x 11 paper.

POSTSCRIPT.PAGEREGIONS [InitVariable]

POSTSCRIPT.PAGEREGIONS is an ALIST mapping pagetypes into paper size and actual imageable area on the page. By default, it knows about LETTER, LEGAL, and NOTE pagetypes, and the corresponding sizes and imageable areas for the Apple Laserwriter. Others can be defined by the user by adding the appropriate entries onto this ALIST.

POSTSCRIPT.PAGETYPE [InitVariable]

POSTSCRIPT.PAGETYPE is used by OPENIMAGESTREAM to lookup the paper size and actual imageable area of the page in POSTSCRIPT.PAGEREGIONS to determine the initial margins. This value can be overridden with the PAGETYPE or PAPERTYPE options in the OPENIMAGESTREAM call. The name of the type of page selected is NOT passed through to the PostScript output.

\POSTSCRIPT.MAX.WILD.FONTSIZE [InitVariable]

\POSTSCRIPT.MAX.WILD.FONTSIZE indicates the maximum point size that should be returned from FONTSAVAILABLE when the SIZE argument is wild (i.e. *). All integer pointsizes from 1 to \POSTSCRIPT.MAX.WILD.FONTSIZE will be indicated as available. The default value is: 72.

POSTSCRIPT.PREFER.LANDSCAPE [InitVariable]

POSTSCRIPT.PREFER.LANDSCAPE indicates if the OPENIMAGESTREAM method should default the orientation of output files to LANDSCAPE. It can have one of three values: NIL, T, or ASK. NIL means prefer portrait orientation output, T means prefer landscape, and ASK says to bring up a menu to ask the preferred orientation if it wasn't explicitly indicated in the OPENIMAGESTREAM call (with the ROTATION option). The default value is: NIL. An item (PS Orientation) is added to the Background Menu to let you change the value of this variable.

POSTSCRIPT.TEXTFILE.LANDSCAPE [InitVariable]

POSTSCRIPT.TEXTFILE.LANDSCAPE indicates if the printing of TEXT files (e.g. LISTFILES, ...) should force the orientation of output files to LANDSCAPE. When it is non-NIL the orientation of output files is forced to LANDSCAPE. (There is no ASK option here.) The default value is: NIL.

POSTSCRIPT.BITMAP.SCALE [InitVariable]

POSTSCRIPT.BITMAP.SCALE specifies an independent scale factor for display of bitmap images (e.g. window hardcopies). Values less than 1 will reduce the image size. (I.e. a value of 0.5 will give a half size bitmap image.) The position of the scaled bitmap will still have the SAME lower-left corner (i.e. the scaled bitmap is not centered in the region of the full size bitmap image). The default value is: 1.

HINT

> Setting POSTSCRIPT.BITMAP.SCALE to 0.96, instead of 1, will give cleaner BITMAP images on a 300 dpi printer. (This corrects for the 72 ppi imagestream *vs*. the 75 dpi printer, using 4x4 device dots per bitmap pixel.) Also, values of 0.24, 0.48 and 0.72, instead of 0.25, 0.5 and 0.75, will also give cleaner images for reduced size output. In general, use integer multiples of 0.24 for a 300 dpi printer.

POSTSCRIPT.TEXTURE.SCALE [InitVariable]

POSTSCRIPT.TEXTURE.SCALE specifies an independent scale for the display of bitmap textures. The value represents the number of device space units per texture unit (bitmap bit). The default value is 4, which represents each bit of the texture as a 4x4 block, so that textures are approximately the same resolution as on the screen (for 300 dpi output devices, such as the Apple Laserwriter).

The PostScript package extends the allowed representations of a texture, beyond 16-bit FIXP and 16x16 bitmap, to ANY square bitmap. (If the bitmap is not square, its longer edge is truncated from the top or right to make it square.) Use this feature with caution, as large bitmap textures, or sizes other than multiples of 16 bits square, require large amounts of storage in the PostScript interpreter (in the printer controller), and can cause limitcheck errors when actually printing.

Anywhere that a texture or color can be used on an imagestream or in the specification of a BRUSH, you can instead give a FLOATP between 0.0 and 1.0 (inclusive) to represent a PostScript halftone gray shade. (0.0 is black and 1.0 is white. Specifically, the value sets the brightness of the shade.) The value you specify will not be range checked, and will be passed directly through to the PostScript setgray operator. (E.g. you can pass 0.33 as the color to DRAWLINE to get a dark gray line with approximately 67% of the pixels in the line black.)

POSTSCRIPT.IMAGESIZEFACTOR [InitVariable]

POSTSCRIPT.IMAGESIZEFACTOR specifies an independent factor to change the overall size of the printed image. This re-sizing affects the entire printed output (specifically, it superimposes its effects upon those of POSTSCRIPT.BITMAP.SCALE and POSTSCRIPT.TEXTURE.SCALE). Values greater than 1 enlarge the printed image, and values less than 1 reduce it. An invalid POSTSCRIPT.IMAGESIZEFACTOR (i.e. not a positive, non-zero number) will use a value of 1. The BITMAPSCALE function for the POSTSCRIPT printer type does NOT consider the POSTSCRIPT.IMAGESIZEFACTOR when determining the scale factor for a bitmap.

MISCELLANEOUS

The SCALE of a PostScript imagestream is 100. This is to allow enough resolution in the width information for fonts to enable TEdit to correctly fill and justify text.

The first time any PostScript imagestream is created (even if only to hardcopy a bitmap or window) the DEFAULTFONT is instantiated (unless a FONTS option was given to the OPENIMAGESTREAM, in which case the initial font for the imagestream will be set to that font, or to the CAR if a list).

The PostScript imagestream method for FILLPOLYGON uses the global variable FILL.WRULE as the default value for the WINDINGNUMBER argument. (This is the same variable which is used by the DISPLAY imagestream method for FILLPOLYGON.)

The PostScript imagestream method for OPENIMAGESTREAM (and, therefore, SEND.FILE.TO.PRINTER), supports an IMAGESIZEFACTOR option to change the size of the printed image. The IMAGESIZEFACTOR re-sizing is combined with the POSTSCRIPT.IMAGESIZEFACTOR to produce an overall re-sizing of the printed image. A HEADING option is also supported to give a running header on each page of output. The value of the HEADING option is printed at the top left of the page, followed by "Page " and the appropriate page number. They are printed in the DEFAULTFONT (unless a FONTS option was given to the OPENIMAGESTREAM, in which case it will be that font, or to the CAR if a list).

The PostScript package is contained in the files: POSTSCRIPTSTREAM.LCOM & PS-SEND.LCOM, with the source in the files: POSTSCRIPTSTREAM & PS-SEND. The module PS-SEND.LCOM is required and will be loaded automatically when POSTSCRIPTSTREAM.LCOM is loaded. It contains the function which is called by SEND.FILE.TO.PRINTER to actually transmit the file to the printer. It is, by its nature, quite site specific, so it is in a separate file to make modifying it for any site relatively simple. System record declarations required to compile POSTSCRIPTSTREAM can be found in EXPORTS.ALL.

I'm pretty sure that the output generated by the PostScript imageops fully conforms to the Adobe Systems Document Structuring Conventions, Version 2.0, January 31, 1987.

Including Other PostScript Operations

If you wish to insert your own specific PostScript operations into a PostScript imagestream, you can do so with the following functions:

(POSTSCRIPT.OUTSTR *STREAM STRING*) [Function]

POSTSCRIPT.OUTSTR outputs a string or value to the imagestream. *STREAM* must be an open PostScript imagestream. *STRING* is the value to output (STRINGP and LITATOM are most efficient, but any value can be output (its PRIN1 pname is used)).

(POSTSCRIPT.PUTCOMMAND *STREAM* $STRING_1$ ... $STRING_n$) [NoSpread Function]

POSTSCRIPT.PUTCOMMAND is more general for sequences of commands and values. It calls POSTSCRIPT.OUTSTR repeatedly to output each of the $STRING_i$ arguments to *STREAM*.

(\POSTSCRIPT.OUTCHARFN *STREAM CHAR*) [Function]

\POSTSCRIPT.OUTCHARFN is used to output the characters forming the text of a PostScript string (e.g. the argument to a show or charpath operator). *STREAM* is the open PostScript imagestream to output to, and *CHAR* is the CHARCODE of the character to output. The ****** (slash), **(** and **)** (parenthesis) characters will be quoted with ******, and characters with ASCII values less than 32 (space) or greater than 126 (tilde) will be output as **\\nnn** (in octal). \POSTSCRIPT.OUTCHARFN will output the **(** character to open the string, if necessary. Use POSTSCRIPT.CLOSESTRING (below) to close the string.

(POSTSCRIPT.CLOSESTRING *STREAM*) [Function]

POSTSCRIPT.CLOSESTRING closes a PostScript string (e.g. the argument to a show or charpath operator). *STREAM* is the open PostScript imagestream. It is important to use POSTSCRIPT.CLOSESTRING to output the **)** character to close the string, because it also clears the stream state flag that indicates that a string is in progress (otherwise, the next POSTSCRIPT.PUTCOMMAND would output the commands to close the string and show it).

Warning

Do not attempt to create a PostScript font larger than about 600 points, as much of Interlisp's font information is stored in SMALLP integers, and too large a font would overflow the font's height, or the width for any of the wider characters. (I know that 600 points is a ridiculously large limit (about 8.3 inches), but I thought I'd better mention it, or someone might try it!)

Changes from the Initial Medley Release

This second Medley release of the PostScript imagestream driver includes some performance enhancements when writing bitmaps to the output, some SUN-specific code (from Will Snow of envos), implementation of the SCALEDBITBLT, DSPROTATE, and DSPTRANSLATE operations, and a lot of performance enhancements (many thanks to Tom Lipkis of Savoir).

Changes from the Lyric Release

The Medley release of this PostScript imagestream driver changed the default value of POSTSCRIPT.TEXTFILE.LANDSCAPE from T to NIL. It also added the support for the HEADING option.

Known Problems/Limitations

The output generated for a PostScript imagestream is rather brute force. It isn't particularly careful to generate the smallest output file for a given sequence of operations. Specifically, it often generates extra end-of-lines between PostScript operator sequences (this has no effect on the printed output, only on the file size).

Using BITMAPs or Functions as BRUSH arguments to the curve drawing functions is not supported, nor is using a non-ROUND BRUSH with DRAWCIRCLE or DRAWELLIPSE.

The implementation of DSPROTATE accepts ROTATION argument values of 0 and 90 (any non-NIL, non-zero value is converted to 90). A value of 0 converts the page orientation to Portrait, and 90 converts the page orientation to Landscape. These conversions perform the translations necessary to keep the clipping region on the page. (This may or may not be the right thing to do, but since DSPROTATE is undocumented in what it should do, this is what the PostScript driver does).

There is no support for NS character sets other than 0, and there is no translation of the character code values from NS encoding to PostScript encoding.

There is no support for color.

\POSTSCRIPT.OUTCHARFN is pretty wimpy in its handling of TAB characters. It just moves to the next multiple of (eight times the average character width of the current font) from the current left margin.

I haven't yet documented how to build the .PSCFONT files from .AFM files for new fonts that become available.

This file describes the UNICODE Lisp Library package.

Contributed by Ron Kaplan, August 2020.

The UNICODE library package defines external file formats that enable Medley to read and write files where 16 bit character codes are represented as UTF8 byte sequences or big-endian UTF16 byte-pairs. It also provides for character codes to be converted (on reading) from Unicode codes to equivalent codes in the Medley-internal Xerox Character Code Standard (XCCS) and (on writing) from XCCS codes to equivalent Unicode codes.

Four external formats are defined when the package is loaded:

```
  :UTF8             codes are represented as UTF8 byte sequences and
XCCS/Unicode character
                        conversion takes place.

  :UTF16BE      codes are represented as 2-byte pairs, with the high order by
appearing
                      first in the file, and characters are converted.
```

The two other external formats translate byte sequences into codes, but do not translate the codes. These allow Medley to see and process characters in their native encoding.

```
  :UTF8-RAW     codes are represented as UTF8 byte sequences, but character
conversion
                      does not take place.

  :UTF16BE-RAW  codes are represented as big-ending 2-byte pairs but there is
no
                conversion.
```

These formats all define the end-of-line convention (mostly for writing) for the external files according to the variable EXTERNALEOL (LF, CR, CRLF), with LF the default.

The external format can be specified as a parameter when a stream is opened:

```
    (OPENSTREAM 'foo.txt 'INPUT 'OLD '((EXTERNALFORMAT :UTF8)))

    (CL:OPEN 'foo.txt :DIRECTION :INPUT :EXTERNAL-FORMAT :UTF8)
```

The function STREAMPROP obtains or changes the external format of an open stream:

```
      (STREAMPROP stream 'EXTERNALFORMAT)  -> :XCCS

      (STREAMPROP stream 'EXTERNALFORMAT :UTF8) -> :XCCS
```

In the latter case, the stream's format is changed to :UTF8 and the previous value is returned, in this example it is Medley's historical default format :XCCS.

Entries can be placed on the variable *DEFAULT-EXTERNALFORMATS* to change the external format that is set by default when a file is opened on a particular device. Loading UNICODE executes

```
      (PUSH *DEFAULT-EXTERNALFORMATS* '(UNIX :UTF8))
```

so that all files opened (by OPENSTREAM, CL:OPEN, etc.) on the UNIX file device will be initialized with :UTF8. Note that the UNIX and DSK file devices reference the same files (although some caution is needed because {UNIX} does not simulate Medley versioning), but the device name in a file name ({UNIX}/Users/... vs. {DSK}/Users/...) selects one or the other. The default setting above applies only to files specified with {UNIX}; a separate default entry for DSK must be established to change its default from :XCCS.

The user can also specify the external format on a per-stream basis by putting a function on the list STREAM-AFTER-OPEN-FNS. After OPENSTREAM opens a stream and just before it is returned to the calling function, the functions on that list are applied in order to arguments STREAM, ACCESS, PARAMETERS. They can examine and/or change the properties of the stream, in particular, by calling STREAMPROP to change the external format from its device default.

The XCCS/Unicode mapping tables are defined by the code-mapping files for particular XCCS character sets. These are typically located in the Library sister directory

```
../Unicode/Xerox/
```

and the variable UNICODEDIRECTORIES is initialized with a globally valid reference to that path. The global reference is constructed by prepending the value of the Unix environment-variable "MEDLEYDIR" to the suffix /Unicode/Xerox/. MEDLEYDIR should be set by the Medley start-up shell script (e.g. /Users/kaplan/local/medley3.5/lispcore/)

The mapping files have conventional names of the form XCCS-<charsetnum>=<charsetname>.TXT, for example, XCCS-0=LATIN.TXT, XCCS-357=SYMBOLS4.TXT. The translations used by the external formats are read from these files by the function

```
(READ-UNICODE-MAPPING FILESPEC NOPRINT NOERROR)
```

where FILESPEC can be a list of files, charset octal strings ("0" "357"), or XCCS charset names (LATIN EXTENDED-LATIN). Reading will be silent if NOPRINT, and the process will not abort if an error occurs and NOERROR. The value is a flat list of the mappings for all the character sets, with elements of the form (XCCC-code Unicode-code).

When UNICODE is loaded the mappings for the character sets specified in the variable DEFAULT-XCCS-CHARSETS are installed. This is initialized to

(LATIN SYMBOLS1 SYMBOLS2 EXTENDED-LATIN FORMS SYMBOLS3 SYMBOLS4 ACCENTED-LATIN GREEK)

but DEFAULT-XCCS-CHARSETS can be set to a different collection before UNICODE is loaded.

The internal translation tables used by the external formats are constructed from a list of correspondence pairs by the function

```
(MAKE-UNICODE-TRANSLATION-TABLES MAPPING [FROM-XCCS-VAR][TO-XCCS-VAR])
```

This returns a list of two arrays (XCCS-to-Unicode Unicode-to-XCCS)containing the relevant translation information organized for rapid access. If the optional from/to-variables arguments are provide, they are the names of variables whose top-level values will be set to these arrays, for convenience. For the external formats defined above, these variables are *XCCSTOUNICODE* and *UNICODETOXCCS*.

The macro

```
(UNICODE.TRANSLATE CODE TRANSLATION-TABLE)
```

is used by the external formats to perform the mappings described by the translation-tables.

The following utilities are provided for lower-level manipulation of codes and strings

```
(XTOUCODE XCCSCODE) -> corresponding Unicode
(UTOXCODE UNICODE) -> corresponding XCCS code
(NUTF8CODEBYTES N) -> number of bytes in the UTF8 representation of N
(NUTF8STRINGBYTES STRING RAWFLG) -> number of UTF8 bytes in the UTF8
      representation of STRING, translating XCCS to Unicode unless
RAWFLG.
(XTOUSTRING XCCSSTRING RAWFLG) -> The string of bytes in the UTF8
representation
      of the characters in XCCSSTRING (= the bytes in its UTF8 file
encoding)
(HEXSTRING N WIDTH) -> the hex string for N, padded to WIDTH
```

The UNICODE file also contains a function for writing a mapping file given a list of mapping pairs. The function

```
(WRITE-TRANSLATION-TABLE MAPPING [INCLUDEDCHARSETS] [FILE])
```

produces one or more mapping files for the mapping-pairs in mapping. If the optional FILE argument is provided, then a single file with that name will be produced and contain all the mappings for all the character sets in MAPPING. If FILE and INCLUDEDCHARSETS are not provided, then all of the mappings will again go to a single file with a composite name XCCS-csn1,csn2,csn3.TXT. Each cs may be a single charset number, or a range of adjacent charset numbers. For example, if the mappings contain entries for characters in charset LATIN, SYMBOLS1, SYMBOLS2, and SYMBOLS3, the file name will be XCCS-0,41-43.TXT.

If INCLUDEDCHARSETS is provided, it specifies possibly a subset of the mappings in MAPPING for which files should be produced. This provides an implicit subsetting capability.

Finally, if FILE is not provided and INCLUDEDCHARSETS is T, then a separate file will be produced for each of the character sets, essentially a way of splitting a collection of character-set mappings into separate canonically named files (e.g. XCCS-357=SYMBOLS4.TXT).

UnixPrint lets you arrange to have hardcopy sent directly to a PostScript printer via a UNIX print command. You can set your default printing host so that it happens automatically.

Installation

Load UNIXPRINT.DFASL. Customize UNIXPRINTCOMMAND.

Then set the two control variables appropriately, as described below:

DEFAULTPRINTINGHOST [Variable]

This is a list of printer names, described in the *Interlisp Reference Manual* (refer to the IRM for a general description). To add a PostScript printer to the list, add an entry in the form (POSTSCRIPT printername). To continue the example above, DEFAULTPRINTINGHOST should have a value like the following:

```
(SETQ DEFAULTPRINTINGHOST
        '(-- (POSTSCRIPT daisy) --))
```

UnixPrinterName [Variable]

A string or symbol, the name of the UNIX printer to which you want output sent. This should be the name that you would give in the lpr command. For example, if you normally print files by entering:

```
lpr -Pdaisy...
```

then you should (SETQ UnixPrinterName "daisy"). If you do not normally specify a printer name, set UnixPrinterName to NIL.

Customization

You can get UNIXPRINT to use lp or lpr by modifying the function UnixPrintCommand. Your site may have a printing program other than lpr. For futher information about printing on your system, please refer to your system manual.

(**UnixPrintCommand** PRINTER COPIES NAME TMPNAME) [function]

Returns a string that is used by /bin/sh in the printing of the postscript code. The arguments are PRINTER, COPIES, NAME and TMPNAME. PRINTER is the name of the printer. COPIES is NIL or a fixp specifying how many copies to print . NAME is the string printed on the banner page of your hardcopy. TMPNAME is the name of the temporary file used to store the postscript code for your job.

A call to `(UnixPrintCommand "daisy" 1 "Erik" "/tmp/foot")` should return something like the string `"/usr/ucb/lpr -Pdaisy -#1 -JErik -r -s /tmp/foot"`.

The source code of the function UnixPrintCommand is supplied with Medley. You are encuraged to write your own versions of this function depending on the site he or she uses. The function `UnixPrintCommand` is included in UNIXPRINT.DFASL. Sources and examples for different versions of `UnixPrintCommand` are included in the file UNIXPRINTCOMMAND.

[This page intentionally left blank]

Before you can print the Medley LispUsers' tedit files you must do the following:

1. If you aren't running in Lispcore you need to load {eris}<tedit>tedithcpy.locm
2. In IL:\ASCIITONS change entry 183 so that it reads 183 instead of what it normally reads.

Doing these two things causes the Envos logo to print correctly. To print some of the LispUsers' modules you must have some functions loaded. The following table shows what you must have loaded to print certain LispUsers' modules:

Module name	File needed to print or display correctly
equation examples	Mathtons.lcom equations.lcom and set il:|EQ.UseNSChars| to T
Doc-objects	docobjects.lcom (loads dateformat.lcom)
hpgl	sketch.lcom

LISPUSERS' MODULES MANUAL

en·vōs

309000
Medley Release
September 1988

Address comments to:
Envos Corporation
1157 San Antonio Road
Mountain View, CA 94043

LISPUSERS' MODULES MANUAL

309000

Medley Release

September, 1988

TABLE OF CONTENTS

[This page intentionally left blank]

{ERIS}<LISPUSERS>LYRIC>LISPUSERS-DEPENDENCIES.TEDIT
Edited 12/16/86 Susana Wessling

This is a list of the LispUsers files that are to be released and their dependencies. Please add new LispUsers files to this list when you write them onto <lispusers>lyric>. If any of the packages on this list have changed or are not to be released, please remove them.

Key:
* internal release only
+ release source AND dcom files
indented: dependent on the above package

```
access.;1
ACE.;1
      ACE-APPLEDEMO.;1
      ACE-BOUNCINGBALL.;1
      ACE-FOUETTE.;1
ACTIVEREGIONS.;1
ACTIVEREGIONS2.;1
AIREGIONS.;1
      AIREGIONS-DEMO.;1
animate.;1
*ARCHIVETOOL.;1
ARITHDECLS.;1
ARRAYSORTER.;1
AUTOSAMEDIR.;1
AUXMENU.;2
BACKGROUND.;1
      +BACKGROUNDDEMO.;1
BACKGROUNDIMAGES.;3
      +BACKGROUND-*.BITMAP
      +BACKGROUND-*.PRESS
BackgroundMenu.;1
BICLOCK.;1
BIZGRAFIX.;1
bltdemo.;1
BMFROMW.;1
BOUNCE.;1
BOUNDARY.;1
BOYERMOORE.;1
      BOYERMOOREDATA.;1
BQUOTE.;1
      BQUOTEGACHA10-C0.DISPLAYFONT
      BQUOTEGACHA12-C0.DISPLAYFONT
      BQUOTEGACHA8-C0.DISPLAYFONT
BTMP.;1
      BTMP-DEBUG.;1
BUGREPORT.;1
CALENDAR.;2
      PROMPTREMINDERS.;1
CCACHE.;1
CD.;1
CD-COMMAND.;1
CHANGEPRINTER.;1
+CIAPROPOS.;1
COLORNNGS.;1
COMHACK.;1
COMPAREDIRECTORIES.;1
COMPARESOURCES.;1
COMPARETEXT.;1
COMPILEFORMSLIST.;1
CONNTITLE.;1
COURIERDEFS.;1
```

```
COURIEREVALSERVE.;1
COURIERIMAGESTREAM.;1
COURIERSERVE.;1
      REMOTEGRAPHER.;1
      REMOTEPSW.;1
CROCK.;1
*CRYPT.;1
DATEFNS.;1
DEDITAUG.;1
DEDITICON.;1 (no .dcom)
DEDITK.;1
DEFAULTICON.;1
DEFAULTSUBITEMFN.;1
DINFOEDIT.;1
DIRECTORYTOOLS.;1
DIRGRAPHER.;1
DIRMENU.;1
DLIONFNKEYS.;1
DOCTOR.;1
DONZ.;1
*DRAWFILE.;1
DSL.;1
dumper.;1
+DUMPLOAD.;1
EDITBG.;1
EDITFONT.;1
+EDITKEYS.;1
EDITRECALL.;1
EMACS.;1
EMACSUSER.;1
EQUATIONFORMS.;1
EQUATIONS.;1
EXEC.;1
+EXECFNS.;1
FACEINVADER.;1
FASTBITMAPBIT.;1
*FILECACHEMSGWINDOW.;1
+FILEOBJ.;1
FILEPERCENT.;1
FILLPRINT.;1
FILLREGION.;1
FINGER.;1
FLAGBROWSER.;1
FLOPPY4.;3
FONTMENU.;1
FULLSCREEN.;1
GKS.;1
GKSEXTERN.;1
GKSINTERN.;1
GKSMATRIX.;1
GLISPA.;1
      GLISPB.;1
      GLISPDWINDOW.;1
      GLISPGEV.;1
      GLISPGEVAUX.;1
      GLISPGEVTYPE.;1
      GLISPR.;1
      GLISPTEST.;1
      GLISPVECTOR.;1
graphcalls.;1
*GREP.;1
HANOI.;1
HASHBUFFER.;1
HASHDATUM.;1
```

```
HEADLINE.;1
HISTMENU.;1
IDEASKETCH.;1
IDLEHAX.;1
IMAGEWRITER.;1
*IMTEDIT.;1
      IMNAME.;1
      IMTOOLS.;1
      IMTRAN.;1
INSPECTCODE-TEDIT.;1
IRISCONSTANTS
      IRISDEMOFNS
      IRISIO
      IRISLIB
      IRISNET
      IRISSTREAM
      IRISVIEW
      LOADIRIS
      SFFONT
      GACHAE.LC1-SF;2
      GACHAE.LC2-SF;2
      GACHAE.NUM-SF;2
      GACHAE.S1-SF;2
      GACHAE.S2-SF;2
      GACHAE.UC1-SF;2
      GACHAE.UC2-SF;2
JARGON.;1
      JARGON.DB
KAL.;1
KEYOBJ.;1
KINETIC.;1
*LAFITEHIGHLIGHT.;1
LCROCK.;1
LIFE.;1
LISTEN.;1
LoadPatches.;1
LOGOCLOCK.;1
LSET.;1
MACWINDOW.;1
MAGNIFIER.;1
magnifyw.;1
*MAILOMAT.;1
MAKEGRAPH.;1
MANAGER.;1
MATHFNS.;1
*MESATOLISP.;1
MOVE-WINDOWS.;1
multimenu.;1
MULTIW.;1
MUSICKEYBOARD.;2
NOTEPAD.;1
      NOTEPAD-CORESTYLES.;1
NQUEENS.;1
PACMAN.;1
PAGEHOLD.;1
PARSER.;1
PARSERG.;1
PATCHUP.;1
PCDAC.;1
PEANO.;1
PERFORMTRAN.;1
PIECE-MENUS.;1
PLAY.;1
PLOT.;2
```

```
      PLOTEXAMPLES.;1
      PLOTOBJECTS.;1
PLURAL.;1
+PQUOTE.;1
PREEMPTIVE.;1
PRESSFROMNS.;1
PRESSTOIP.;1
      DPRESS.;1
PRINTERMENU.;1
Proofreader.;1
      ANALYZER.;1
      SpellingArray.;1
PULLDOWNMENUS.;1
+QIX.;1
*READBRUSH.;1
RECORDPRINT.;1
REGION.;1
RESETMACROS.;1
ROTATEBM.;1
*RPC.;1
      RPC-EXAMPLE.;1
      RPC-EXAMPLECLIENT.;1
      RPC-EXAMPLESERVER.;1
      RPC-EXAMPLEUSER.;1
      RPC-LUPINE.;1
      RPCEVAL.;1
      RPCEVALCLIENT.;1
      RPCEVALSERVER.;1
*SAMPLER.;1
SERVERSTATUS.;1
SETDEFAULTPRINTER.;1
SETF.;1
SHOW.;1
SIGNAL.;1
SINGLEFILEINDEX.;1
SLIDEPROJECTOR.;1
SNAPSCROLL.;1
SOLITAIRE.;1
SPACEWINDOW.;1
sprint.;1
STARBG.;1
STOCKICONS.;1
STYLESHEET.;1
+SUPERMENUEDIT.;1
+SUPERMENUS.;1
SYSTAT.;1
+TEDITKEY.;1
THERMOMETER.;1
      THERMOMETERDEMO.;1
TILEDEDIT.;1
TIMEPANEL.;1
+TINYTIDY.;1
TMENU.;1
TOGMENU.;1
TRACEIN.;1
TRAJECTORY-FOLLOWER.;1
      TRANSOR.;1
TRUEHAX.;1
TSET.;1
TTY.;1
ttyio.;1
TURING.;1
TWOD.;1
TWODGRAPHICS.;1
```

UNBOXEDOPS.;1
utilisoprs.;1
VMEMSTATE.;1
VSTATS.;1
+WAM.;1
WDWHACKS.;1
WINK.;1
WINNER.;1
WORM.;1
+YAPFF.;1

Other TEDIT files: [on {eris}<lisp>koto>lispusers> or {eris}<lispusers>koto>]

Release-intro.tedit	intro for released doc’s
Release-rules.tedit	rules for released doc’s
Internal-intro.tedit	intro for internal doc’s
Internal-rules.tedit	rules for internal doc’s
blankpage.tedit	blank page for printing screwups...
documentationtemplate.tedit	long template for doc’s
easytemplate.tedit	short template for doc’s
lispusers-rules.tedit	old LU rules
RELEASE-INFO.tedit	This document
LISPUSERS.TEDIT	Short package summaries

Subject: Re: Lispusers packages (updates)
In-reply-to: bloomberg.pa’s message of 16 Dec 86 12:38 PST
To: bloomberg.pa
cc: LispUsers^.x
Reply-to: Wessling.pa
Format: TEdit

I received this inquiry today:

Date: 16 Dec 86 12:38 PST
From: bloomberg.pa
Subject: Lispusers packages (updates)
To: wessling.pa
cc: bloomberg.pa

When I replace text or documentation of a Lispusers package with new versions, should I message you so that you can do the appropriate thing(s) vis-a-vis phylum?

--Dan

...and I thought this would be a good time to just give an update on how to handle the lispusers packages.

--- Directories:

{eris}<lisp>koto>lispusers> is where the frozen, released packages for Koto were put.
{eris}<lispusers>koto> is where the Koto packages that have been changed since release should be put.
{eris}<lispusers>lispcore> is where new packages that run in LispCore and updates of old packages that are updated to Lispcore should be kept.

--- Creating new packages:

if you create a new package that runs in LispCore, i.e. to be release with Lyric, write it onto {eris}<lispusers>lispcore>.

--- Editing old packages:

if you want to change a package that was released with Koto, make changes to the latest version (if it hasn’t changed since release this is on {eris}<lisp>koto>lispusers> or if it has been updated, it should have been put on {eris}<lispusers>koto>) and copy it to:
{eris}<lispusers>koto> if it **only** still runs in Koto
{eris}<lispusers>Lispcore> if you **know** it runs in Lyric.

--- What should be included:

No package will be released from LispUsers without:
- Source file
- Compiled file

- Documentation

Documentation can be made through the template ({eris}<lispusers>LispCore>easytemplate.tedit) or by using the new lispusers package, DOCUMENT (on {eris}<lispusers>koto>).

--- for more information:

{eris}<lispusers>lispcore>LispUsers-rules.tedit, LispUsers-Summary.tedit, LispUsers-dependencies.tedit. I will store this on {eris}<lispusers>lispcore>lispusers-info-msg.tedit.

--- Nice things to do:

When you create a new package, please:
- send a note to LispUsers^ announcing it.
- add it to LispUsers-summary.tedit (a list of all the lispusers packages)
- add it to LispUsers-dependencies.tedit, which is a list of all the lispusers packages and which files they need to run.
- Edit LispUsers-dependencies.tedit to reflect the current state of Lispcore lispusers files (right now it is just the info from Koto).

If you created a package for Koto, please try to update it to Lispcore and store it onto <lispusers>lispcore>. If the package is un-updatable or useless in LispCore, please take it off of the aforementioned lists and/or message me about it.

--- and to answer the question that started all this:

Just put your packages on the directories mentioned. Any copying will be done.

Thank you very much for your cooperation....

Susana...

Date: 23 Jan 87 17:21 PST
From: Wessling.pa
Subject: {eris}<lispusers>lyric> exists!
To: LispUsers^.x
Really-from: The Tired Floppy Lady (let's all go home... it's Friday...)
Reply-to: Wessling.pa

NOTICE: If you are either a user or a creator/maintainer or ANY LispUsers module, please please please do not delete this message. I am sick of resending information...

All files that existed on {eris}<lispusers>lispcore> as of NOW have been moved to {eris}<LispUsers>Lyric>.

From now on, follow these rules for storing new or changed LispUsers modules:

- If the module was created or changed in Koto, please store it under {eris}<lispusers>koto>.

- If the module was created or changed in the {eris}<lyric>basics> sysout, please store it on {eris}<lispusers>lyric>.

- If the module was created or changed in the {eris}<lispcore>next> sysout, please store it on {eris}<lispusers>lispcore>.

Though this may seem a bit complicated, it will help us to keep track of which modules are assured of working where.

****
RULES:

AND NOW... I am once again going to remind you of how you can make our lives a bit easier when interacting with LispUsers:

All modules that are stored on ANY of the aforementioned directories MUST have:

- source file
- compiled file (.LCOM or .DFASL for Lyric, .DCOM for Koto)
- documentation

If you do not store any of the above with your module, it WILL NOT be released.

In writing documentation, you can use EasyTemplate.tedit (stored on any/all LU directories) to make life easier. For a long version, see DocumentationTemplate.tedit. Please please please note in your documentation:

- any dependencies between modules
- if the module is for internal use only

For further and more detailed rules, see Lispusers-rules.tedit.

If you are nice, I would appreciate your updating all of the <lispusers>lispusers* files. These files contain information about releasable modules that is very important. You should look into these files if
- your module requires other modules to be loaded with it.
- your module is for internal release only
- your module is new

****
QUESTIONS/REMINDERS:

It has been brought to my attention that there are some modules that are stored on {eris}<lispusers>lyric> that are definitely NOT lyric compatible. Please do not do this. It will not make anyone’s job easier. Such modules should go on <lispusers>koto>. Please move them if you are responsible for such horrific actions!

If you are responsible for or just have a vested interest in some module that is in <lispusers>koto> but has not been updated, please see to getting it updated and put into <lyric> or <lispcore>. If it is not done, it will not go out with Lyric.

For those who are anxiously waiting for an NS LispUsers directory... I’m working on it. I will send out further information if it happens.

****
THAT’S ALL FOLKS...

Sorry for the long message, but these things just had to be said....

Susana

—*End of message*—

LISP USERS' RULES

This document describes the rules and procedures for Xerox Lisp "Lisp Users'" modules. This document is mainly for Lisp Users' module writers, but users should also understand the rules. The deadline for Medley Lisp Users' module submissions is August 5, 1988.

DEVELOPING A LISP USERS' MODULE

A Lisp Users' module is a useful program made available to the general Xerox Lisp programming community. Neither the author nor the custodian of the Lisp Users' directory imputes any warranty of suitability or responsibility for errors.

Lisp Users' modules should be easily distinguishable from released library modules. In particular, this means that a Lisp Users' module may not have the same name as a Library module and should be visibly different. Lisp Users' modules derived from released software should be announced to the public only after communicating with the organization responsible for that released software.

Testing is important. If you make significant changes to a Lisp Users' module, enlist developers at your site as alpha testers. Avoid having to rerelease a package within hours of its announcement because of fatal bugs. A Lisp Users' module is not shoddy software; it is software made available outside the regular release channels.

Try not to flood your user community with a constant flow of new versions (or messages) so your messages can be appreciated rather than discarded without reading. If your module is undergoing continual changes, adopt a release strategy of regularly spaced, well tested releases. Your users will thank you.

LISP USERS' MODULE OWNERSHIP

A module submitted for Lisp Users' remains the "property" of the submitter. Others may not make changes, except for their own private use, without negotiating with the owner (who may already be making similar or incompatible changes).

As the owner of a module, you are not required to fix bugs, but if not, you must be willing to transfer ownership (permanently) to someone who volunteers to fix them. Ownership may pass back and forth among several people as long as they agree.

Like all software developed at Xerox, Lisp Users' modules are officially the property of Xerox. You should run with the COPYRIGHT option set to produce an appropriate Xerox copyright in the source.

A Lisp Users' module may become so useful that it becomes part of a standard Xerox Lisp release and is thereafter supported. Ownership then passes to the Xerox product organization.

SUBMITTING LISP USERS' MODULES

If you are not an internal user, you should submit your new module to Xerox either through e-mail or on a floppy. External users should make sure that they include all relevant information, such as documentation containing an e-mail or US mail address where he/she can be reached. External users are also held responsible for the support of their modules.

SUBMITTING FILES TO LISP USERS'

As with released software, it is important to submit not just the resulting product, but all the files needed to build and maintain a Lisp Users' module:

1. the runnable compiled file (.LCOM or .dfasl)
2. documentation describing it, following the set formatting rules (see below)
3. a source file that can be released (optional)
4. data files, if needed

Packages submitted once are released once. Do not assume that a package submitted in one release will be automatically released in subsequent releases.

DOCUMENTATION

Documentation, essential to any module being used effectively, should be put on the submitted floppy. No modules will be released without documentation. Documentation can be as simple as a paragraph describing what the module does and how to use it, or it can be as extensive as a dozen-page user manual. All modules should have a file with a .TEDIT extension. Formatting should be done according to the rules outlined in the appropriate (standalone or networked user) Lisp Users' Template. All users, external users included, should follow the Lisp Users' Template rules. If the documentation is large and formatting time consuming, you can also produce an interpress file (with the .ip extension), as well as submitting a .TEdit file. (Be sure to update the interpress file if you update the documentation!) Documentation should include the full electronic mail address of the submitter.

COMPATIBILITY WITH LISP USERS'

Any submitted Lisp Users' files should be compilable in a "vanilla" environment. The file itself should load in any auxiliary modules under a suitable (DECLARE: EVAL@COMPILE --) when necessary.

Thanks for your cooperation.

Lispusers Summary

A one-line description of (most) LispUsers packages.

All files listed here should be in {eris}<lispusers>lispcore>. If you want to look at the list for Koto files, see <lispusers>koto>, the same file. Please add any new files you create to this list.

---NO TEDIT FILE means that no documentation is needed.
---UNSUPPORTED means that there is no author or the author is unreachable.
---an asterisk before a file means its part of the previous package.
---INTERNAL means that it is not for release outside of Xerox
---SOURCE means that you load the source & not the DCOM
---Two dashes (--) before it means that it exists in Koto but hasn't been updated for lyric yet.

Module	Comments
--ACCESS	
--Ace	Animation Compiler; includes graphics editor.
--* ACE-EDIT, ACE-Main, ACE-Prim, ACE-AppleDemo (Source), Ace-BouncingBall (Source), ACE-Fouette (Source)	
--ActiveRegions	Mouse sensitive regions in a window.
--ActiveRegions2	Rewrite of ActiveRegions
--AIRegions	
--* AIREGIONS-DEMO (Source)	
--Analyzer	Part of proofreader
--Animate	Smoothly move arrow, finger around on screen.
--ArithDecls	For use with decl
--ArchiveTool	(INTERNAL) Cedar archive system interface
--ArraySorter	
--AutoSameDir	Put sources back where you got them at MAKEFILE
--Auxmenu	Useful middle-button menu in background
--Background	BITBLT to screen background or obscured windows.
--* BACKGROUNDDEMO	
--BackgroundImages	screen backgrounds, including rhine, two dollar bill, castle
BackgroundMenu	Useful menu in the background
BiClock	Swedish clock
BITMAPFNS	
--BizGrafix	Pie & bar chart, line graph creation
--BLTDemo	graphics demo/idle hack
--BMFromW	bitmap from window
--Bounce	Idle hack with lines
--Boundary	
BoyerMoore	The Boyer-Moore Theorem Prover
--* BoyerMooreData (Source)	
--BQuote	Yet another backquote macro (like Common Lisp's)
BSEARCH	
--BTMP	Basic Text Macro Processor
--* BTMP-Debug	
--BugReport	
--Calendar	calendar/appointment-reminder program
--CCache	keep files on local disk, like FILECACHE but less automatic
CD	Connect-to-directory command
--CD-Command	Connect-to-directory command
--Changeprinter	Interface for dealing with printers
CHATSERVER	
CHECKSET	
--CIApropos	Case Independent apropos

CIRCLPRINT	
--ColorNNGS	For use with busmaster and Number 9 Graphics Card on 1108
--ComHack	Comments in if and fors
COMMENTSTRINGS	
COMMON-MAKEFILE	Creates plain-text exportable Common Lisp source code files
COMMWINDOW	
--CompareDirectories	What files are different on two directories
CompareSources	Compare two lisp files
--CompareText	Compare two text files
COMPILEBANG	
--CompileFormsList	
--ConnTitle	Show where I'm connected
--CourierDefs	Implementation of XNS protocol for remote procedure call
--CourierEvalServe	EVAL server using courier
--CourierImageStream	Image stream server using courier
CourierServe	courier server
--Crock	Analog clock
--Crypt	(INTERNAL)
--Datefns	
--DEditAug	
--DEditHardcopy	(NEW) hardcopy of Dedit window prettyprints whole function
--DEditIcon	An icon for Dedit
DEditK	New DEdit menu with common combo's
--DefaultIcon	new default icon for shrunk windows
--DefaultSubitemFN	
DES	
DIGEST	
DInfoEdit	
--DirectoryTools	
--Dirgrapher	Shows a graph of directory structure
DLionFNKeys	Dandlion keys (center/bold/etc) for 1132 users
DMSG	
--Doctor	Infamous Eliza program (computer shrink)
--Donz	talking windows
--DPress	(INTERNAL) take apart press files
DSKTEST	
--DSL	Digital Speech Lab, uses Busmaster speech analysis
--Dumper	(INTERNAL) Alto Exec DUMP files
--DumpLoad	Alto Exec DUMP files
--EditBG	Edit background & background border shade
--EDITFONT	Create & edit display fonts
--EditKeys	Dandlion keys (center/bold/etc) for 1132 users
--EditRecall	
EMACS	EMACS commands on top of tedit
--EMACSUser	Use EMACS as your programming environment
--EquationForms	TEdit file=EquationEditor.tedit; for editing equations
--Equations	TEdit file=EquationProgram; programmatic interface
--Exec	Extra exec windows (like Listen)
--ExecFNS	INTERNAL
--FaceInvader	A game
--FastBitmapBit	
FILECACHE	
FILECACHE-BROWSER	
FILECACHE-DECLS	
FILECACHE-HOSTUP	
FILECACHE-SCAVENGE	
FILECACHEMSGWINDOW	

--FileObj	Files as image objects
--FilePercent	Lanning
FILEWATCH	
--FillPrint	
--FillRegion	
--Finger	Who else is running Finger on the net?
--FlagBrowser	
--FontMenu	
--FullScreen	
--GKS	Graphics Kernel Standard implementation
--* GKSExtern	
--* GKSIntern	
--* GKSMatrix	
--GLISP	Compiler for GLisp, an object-oriened Lisp language
--* GLISPA	
--* GLISPB	
--* GLISPDWINDOW	
--* GLISPGEV	
--* GLISPGEVAUX	
--* GLISPGEVTYPE	
--* GLISPR	
--* GLISPTEST.LSP	
--* GLISPVECTOR.LSP	
--* GLISP.tty	
--GraphCalls	Graph calls from interpreted,compiled code
--GREP	NEW. Search file(s) for strings, e.g. phone book.
--Hanoi	Graphics demo/idle hack
--HashBuffer	
--HashDatum	
--Headline	Big titles on the screen
--HistMenu	History list as a menu
--IdeaSketch	
IdleHax	collection of idle hacks
--IdleSwap	(NEW) idle hack
--ImageWriter	output to Apple image writer
--IMName	(INTERNAL) edit Interlisp Manual
--IMTedit	(INTERNAL) convert Interlisp Manual to TEdit
--IMTools	(INTERNAL) Tools for dealing with Interlisp Manual
--IMTran	(INTERNAL) Translate Interlisp Manual
--InspectCode-TEdit	
INVISIBLEWINDOW	
--Jargon	definitions from the hacker’s dictionary
--Kal	b/w or color kaleidoscope (also in Idlehax)
--? KeyboardTool	
Keyobj	key image-objects
--Kinetic	Graphics demo/idle hack
--LCrock	Clock in the logo window
Life	Conway’s game of life, as an Idle hack
LISPXCONVERT	
--Listen	Lisp Executives from the background menu
--LoadPatches	
--LogoClock	Another clock in the logo window
--LSet	Lists as sets
MacWindow	Shrink & expand windows like a MAC
--Magnifier	Magnify areas of the screen
--MagnifyW	
--MailoMat	INTERNAL
--MakeGraph	help for Grapher users making graphis
Manager	Window/menu file package interface
MANDELBROT	
--MathFNs	trig, complex functions, constants
MERGE-FILEGEN	

--Move-Windows	
--MTP	(INTERNAL) Mail Transfer Protocol for Lafite<->Tops-20
--MultiMenu	Attached menus in groups
--MultiW	Heirarchical window environment
--MusicKeyboard	
--Notepad	Graphics paint program
--* NOTEPAD-CORESTYLES (SOURCE)	
NOVAFONT	
NSCHATSERVER	
NSDISPLAYSIZES	
NSMAINTAIN	
NSREADERPATCH	
--NSthasize	(INTERNAL NEW) Convert GV Distribution list to NS
--NQueens	Graphics demo
--Pacman	Game
--Pac-Man-Idle	(NEW) Idle hack
PageHold	Changes "window full" behavior on scrollers
--Parser	Parser generator for making new parsers
--Patchup	
--PCDAC	A-to-D and D-to-A using BusMaster and PC boards
--Peano	Graphics demo
--Performtran	add clisp word to record package
--Piece-Menus	
PLANETS	
--Play	Tunes on 1108/1186 beeper
--Plot	Making plots
--* PlotExamples	Some samples
--* PlotObjects	Plots as image objects in documents
--Plural	Plural of words
--PQuote	Prettyprint (QUOTE x) as 'x.
--Preemptive	make scheduler preemptive (caveats)
--PressFromNS	
--PressToIP	
--PrinterMenu	
--PromptReminders	reminders at a given time, used by Calendar
ProofReader	Spelling checker in Tedit
PUPCHATSERVER	
--* SpellingArray	
--PullDownMenus	
--QIX	Shrager
--RecordPrint	
--Region	
--RemoteGrapher	Grapher over XNS connection on another machine
--RemotePSW	Someone elses Process Status WIndow on your screen
--ResetMacros	
--RotateBM	Rotate bitmaps
--RPC	(Internal) Cedar-style PUP based Remote Procedure Call
--* RPC-Example	
--* RPC-ExampleClient	
--* RPC-ExampleUser	
--* RPC-Lupine	
--* RPCEVAL	
--* RPCEVALCLIENT	
--* RPCEVALSERVER	
RS232CHATSERVER	
--Sampler	Graphics demo
SCREENPAPER	
--ServerStatus	
--SetDefaultPrinter	
--SetF	Common Lisp style SETF

--Show	
--Signal	Mesa-style signals
SingleFileIndex	Add index to ListFiles output
--SlideProjector	Cycle thru tedit file of "slides"
--SnapScroll	Scrollable "snap" windows
--Solitaire	Graphics demo/Idle hack
--SpaceWindow	space allocation use in a window
--SPrint	
--StarBG	Stars in the background/Idle hack
--? StockIcons	
--StyleSheet	create block of menus
--SuperMenus	
--* SuperMenuEdit	
--Systat	control-T puts up window w/graphic display
TCPCHATSERVER	
TEditKey	New TEdit commands as various meta-characters
--Thermometer	
--* THERMOMETERDEMO	
--TileDEdit	Dedit windows place themselves so they don't overlap
--TimePanel	1108 Maintenance panel => clock
TinyTidy	Move icons over to edge of screen
--TMenu	menus that stuff input buffer, pull down menus
--TogMenu	
--TraceIn	stepper/tracer for debugging
--Trajectory-Follower	animation of following a trajectory
--Transor	
--* TSet	
--TrueHax	
--TTY	
--TTYIO	
--Turing	Turing machine simulator
--TwoD	Two dimensional graphics package
--TwoDGraphics	Two dimensional graphics package
UnboxedOps	Unboxed arithmetic for dandetiger's
UPCSTATS	
--UtilISOpr	Additional iterative operators
--VMemState	Turn on/off VMEM.PURE.STATE
--VStats	Show time, space used
--WAM	Window Action Menu
--WDWHacks	
WHO-LINE	
WHOCALLS	
--Wink	Graphics demo
--Winner	
--Worm	Idle hack
--Yapff	Yet Another Page Full Function

>>MODULE NAME<<

By: >>Your Name<< (>>Your net address<<)

Uses: >>Other modules necessary to run this one<<

>>Type INTERNAL here if the file is for Internal Use Only<<

This document last edited on >>DATE<<.

INTRODUCTION

>>This paragraph should be replaced by an overview of your module. The attached Lisp Users' Template explains the documentation conventions to be used for each Lisp Users' modules.<<

MODULE EXPLANATIONS

>>Functions, Variables, and Lisp Code Examples<<

It is usual to first give the name of a function, then describe its purpose and each of its arguments. When the name of a function is first given, it is set off like this:

(IMAGEFNSCREATE *DISPLAYFN IMAGEBOXFN PUTFN GETFN COPYFN*) [Function]

The function name is in 10-point regular Modern, all caps. Arguments are in 10-point italic Modern, in all caps, mixed case, or lowercase, as they appear in the system. Variables look like functions, except that the word ''Variable,'' enclosed in **square brackets**, follows the variable name. Please note that these are the characters [], not the parentheses

```
This is an example of code. It is in 10-point Terminal font.
```

Function names, commands, file names, and the like are in 10-point modern.

Be sure to include the following information in any module explanations:

• any file dependencies

• definitions of all arguments

• module, function variable, etc. limitations

• a liberal number of examples for all functions, variables, etc.

LISP USERS' TEMPLATE

Updated by: Melissa Biggs (Biggs.PA @ Xerox.COM)

This document provides a template and instructions for formatting the Lisp Users' module documentation. This template applies primarily to standalone workstation users. Using the Lisp Library module TEdit, and this document, you should be able to create a standard Lisp Users' module for the Lisp Users' manual. This document gives you the written specifications for formatting your document. The specifications are given in the order in which you would most likely use them to format a document, with the basic text and margins described first, then the various levels of headings, then special elements such as page numbers.

RULES FOR CONTENT

Documentation should always include the name of the module, the name of the author (and Xerox, Arpanet, CSNET or other electronic mailing address, when available—otherwise US mail address), the names of all other Lisp Users' modules required, the names of all files which are part of the module (data files, other Lisp files, etc.), and enough detail to allow someone to effectively use it. A sample Lisp Users' template appears at the end of this document.

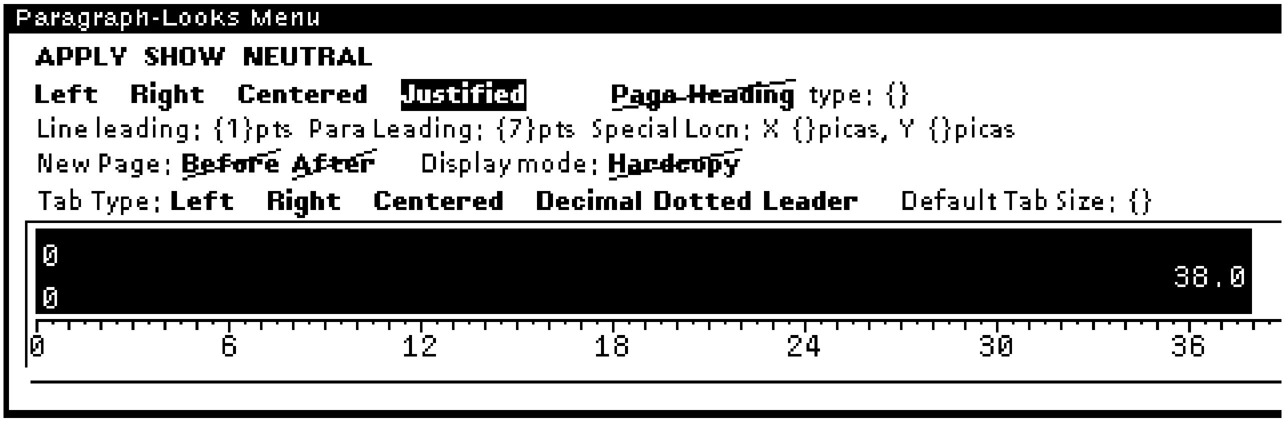
BASIC SPECIFICATIONS

It is wise to apply the specifications for the body text, headings, functions, variables, and page numbers as you write the document.

Font, Type Size, Leading, and Margins

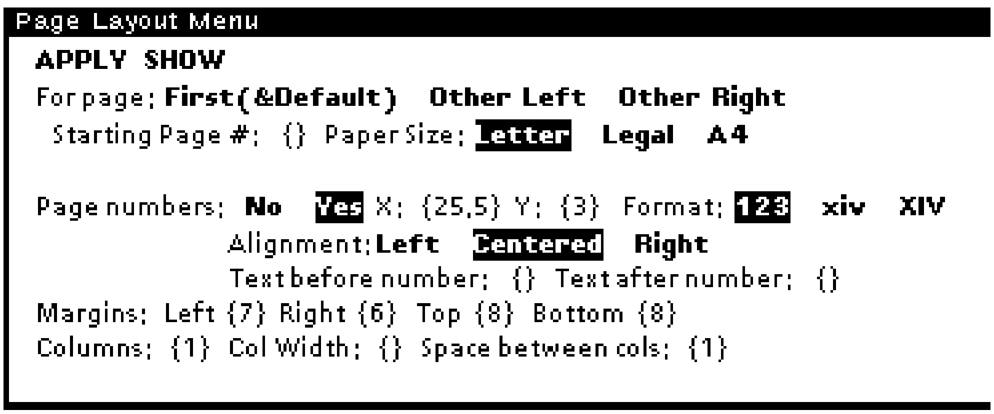
For the text, choose a 10-point Modern font and Apply it to the appropriate text using the Character Looks menu.

Then, in the Paragraph Looks menu, set the leading, the spacing between lines of type; the justification; and the left and right margin settings. Set line leading to 1 point and paragraph leading to 7 points. Apply all paragraph looks to the appropriate text.



(Because 7 points paragraph leading is all you need, you should use only one carriage return between paragraphs when typing in text.)

Finally, in the Page Layout menu, set the left margin to 7 picas, the right margin to 6, and the top and bottom margins to 8. Apply these to all types of pages (first, other left , and other right).

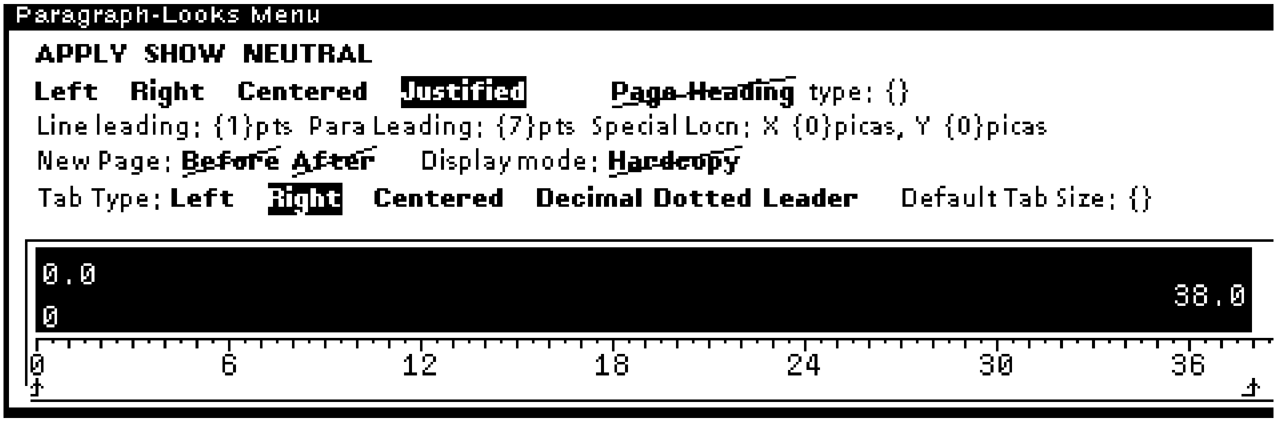


Functions, Variables, and Lisp Code Examples

It is usual to first give the name of a function, then describe its purpose and each of its arguments. When the name of a function is first given, it is set off like this:

(IMAGEFNSCREATE *DISPLAYFN IMAGEBOXFN PUTFN GETFN COPYFN*) [Function]

The Paragraph Looks menu for a function is set up like this:

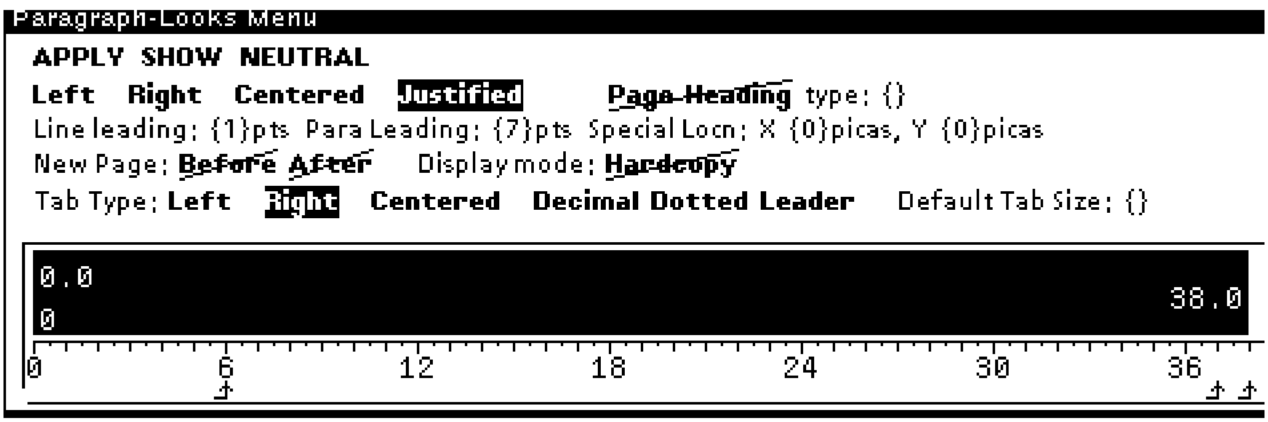


The function name is in 10-point regular Modern, all caps. Arguments are in 10-point italic Modern, in all caps, mixed case, or lowercase, as they appear in the system.

If the function description is more than one line long, the runover arguments should be indented under the function name and the word [Function] placed on the last line of the argument list, like this:

(MAKE-ARRAY *INDICESLIST &KEY :ELEMENT-TYPE:ˆ*
INITIAL-ELEMENT :INITIAL-CONTENTS :ADJUSTABLE:ˆ
FILL-POINTER :DISPLACED-TO :DISPLACED-INDEX-OFFSET) [Function]

The Paragraph Looks menu should be set as follows to produce this type of argument format:



Long function descriptions should have two points leading between lines. To space them correctly, hold down the meta key when typing the carriage returns so that TEdit breaks the lines without creating separate paragraphs.

Parentheses are in 10-point regular Modern type. The type of definition is in 10-point Modern, caps and lowercase, enclosed in square brackets, and flushed right with a right tab set to 38.0. Note that tabs are used to indent the second and third lines of arguments.

Variables look like functions, except that the word "Variable," enclosed in square brackets, follows the variable name.

Examples of code should be in 10-point Terminal font (but not function names, commands, file names, and the like). If 10-point Terminal is not available on your printer, use 8-point Terminal. Code examples should have two points line leading and no paragraph leading.

Quotation Marks, Bullets, and Dashes

We recommend using TEdit's "expanded abbreviations" to produce professional-looking quotation marks, bullets, em-dashes—used to separate text phrases—and en-dashes (used to indicate inclusive numbers, as in "pages 3–6").

· To produce a bullet, type a lowercase b, select it, and type Control-X.

· To produce an em-dash, type a lowercase m, select it, and type Control-X.

· To produce an en-dash, type a lowercase n, select it, and type Control-X.

HEADS

There are four levels of heads in the Lisp Users' documentation: chapter (level 1) heads, level 2, level 3, and level 4 heads.

Note: A head that falls at the bottom of a page (a "widow") is undesirable. You eliminate a widow by selecting it, then applying the Before option of the New Page command in the Paragraph Looks menu.

The Chapter Head

The chapter head appears at the beginning of the document and identifies it. The heading "Lisp Users' Template," above, is a correctly formatted Lisp Users' chapter head.

When submitting a Lisp Users' module on floppy disk format the chapter head as follows:

Module Name: your Lisp Users' Module (all caps, 12-point bold Modern)

Your name and ARPANET address (if you have one) in 10-point Modern

The Level 2 Head

Level 2 heads identify major sections of a document. The level 2 heads for the Lisp Users' documentation are in 10-point bold Modern, all caps.

The Level 3 Head

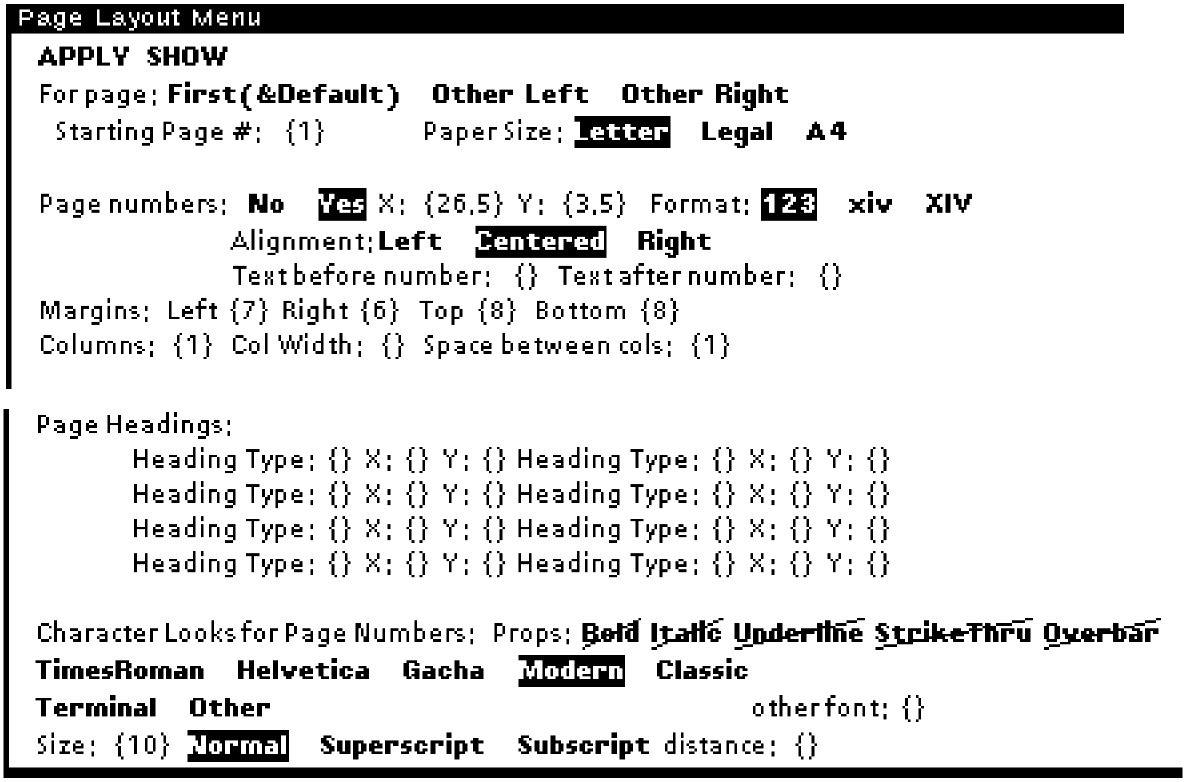
Level 3 heads identify subsections of a document. For the Lisp Users' manual, they are in 10-point bold Modern, caps and lowercase.

The Level 4 Head

Level 4 heads identify the lowest level of subsection in the Lisp Users' documentation. They are in 10-point regular Modern, caps and lowercase, underlined.

PAGE NUMBERS

Page numbers are specified and applied in the Page Layout menu. First, specify the alignment of the page numbers to be centered, with the *X* position being 26.5 picas and the *Y* position 3.5. Then specify the character looks to be 10-point regular Modern. Finally, apply the page numbers to the First(&Default) pages.



After you submit your document, XEROX AIS will add running heads, put in additional formatting, and provide final page numbering to assemble it into the Lisp Users’ manual.

MANUAL TEMPLATE

The following page has a sample template for the information required in a Lisp Users’ module. Use this template to produce your module documentation.

Module Name:

>>MODULE NAME<<

By: >>Your Name<< (>>Your net address<<)

Uses: >>Other modules necessary to run this one<<

>>Type INTERNAL here if the file is for Internal Use Only<<

This document last edited on >>DATE<<.

INTRODUCTION

>>This paragraph should be replaced by an overview of your module. The information on the previous pages explains the documentation conventions to be used for each Lisp Users’ module.<<

MODULE EXPLANATIONS

>>Functions, Variables, and Lisp Code Examples<<

It is usual to first give the name of a function, then describe its purpose and each of its arguments.

Module explanations may have several level headings.

Be sure to include the following information in any module explanations:

• any file dependencies

• definitions of all arguments

• module, function variable, etc. limitations

• a liberal number of examples for all functions, variables, etc.

LISP USERS' TEMPLATE

Updated by: Melissa Biggs (Biggs.PA @ Xerox.COM)

This document provides a template and instructions for formatting the Lisp Users' module documentation. This template applies primarily to standalone workstation users. Using the Lisp Library module TEdit, and this document, you should be able to create a standard Lisp Users' module for the Lisp Users' manual. This document gives you the written specifications for formatting your document. The specifications are given in the order in which you would most likely use them to format a document, with the basic text and margins described first, then the various levels of headings, then special elements such as page numbers.

RULES FOR CONTENT

Documentation should always include the name of the module, the name of the author (and Xerox, Arpanet, CSNET or other electronic mailing address, when available—otherwise US mail address), the names of all other Lisp Users' modules required, the names of all files which are part of the module (data files, other Lisp files, etc.), and enough detail to allow someone to effectively use it**.** A sample Lisp Users' template appears at the end of this document.

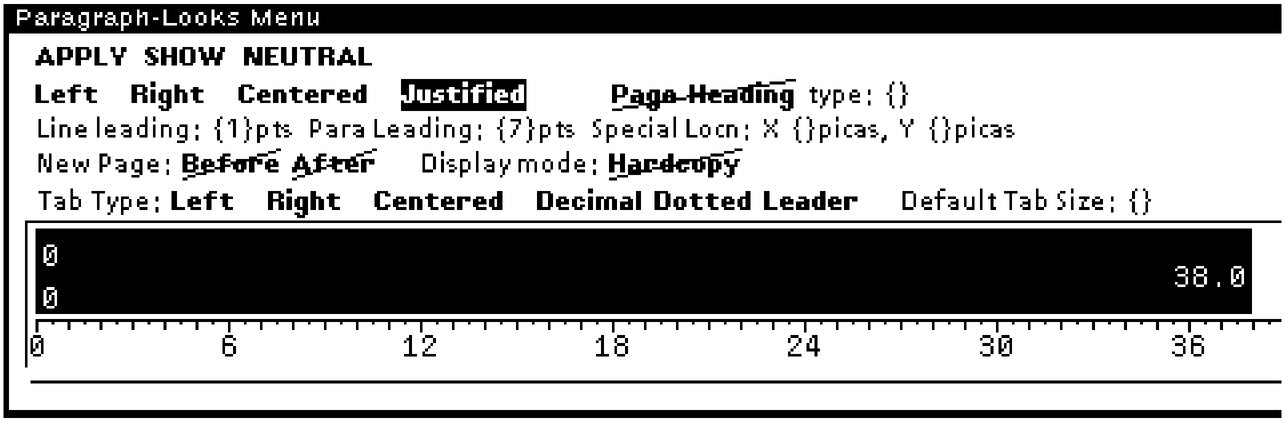
BASIC SPECIFICATIONS

It is wise to apply the specifications for the body text, headings, functions, variables, and page numbers as you write the document.

Font, Type Size, Leading, and Margins

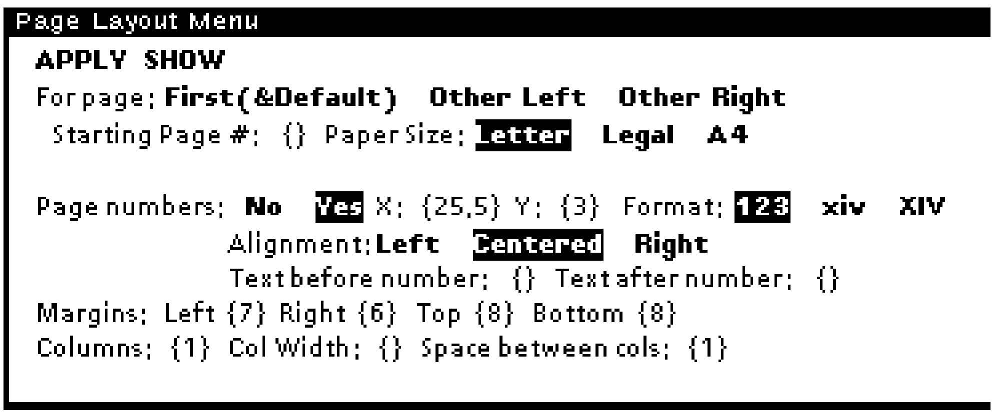
For the text, choose a 10-point Modern font and Apply it to the appropriate text using the Character Looks menu.

Then, in the Paragraph Looks menu, set the leading, the spacing between lines of type; the justification; and the left and right margin settings. Set line leading to 1 point and paragraph leading to 7 points. Apply all paragraph looks to the appropriate text.



(Because 7 points paragraph leading is all you need, you should use only one carriage return between paragraphs when typing in text.)

Finally, in the Page Layout menu, set the left margin to 7 picas, the right margin to 6, and the top and bottom margins to 8. Apply these to all types of pages (first, other left , and other right).

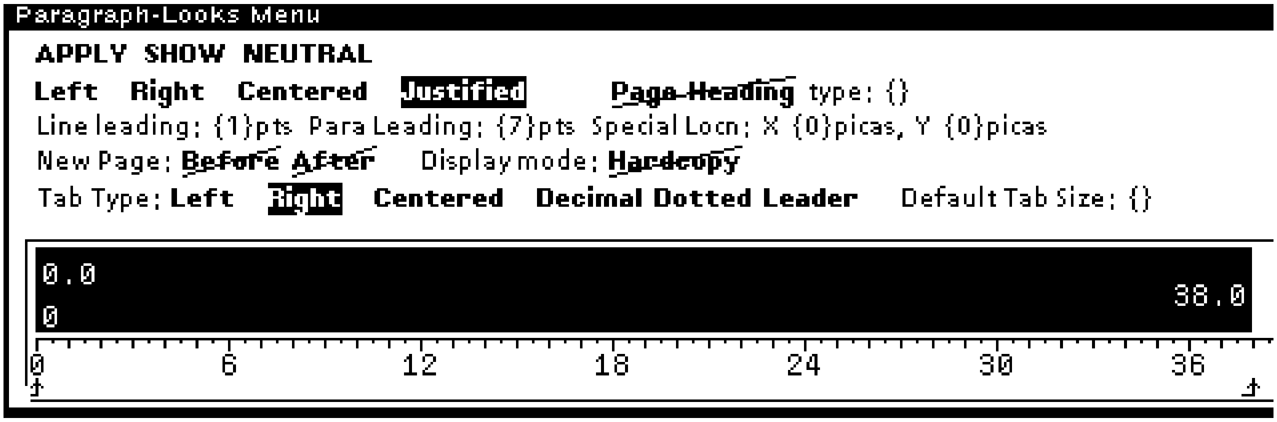


Functions, Variables, and Lisp Code Examples

It is usual to first give the name of a function, then describe its purpose and each of its arguments. When the name of a function is first given, it is set off like this:

(IMAGEFNSCREATE *DISPLAYFN IMAGEBOXFN PUTFN GETFN COPYFN*) [Function]

The Paragraph Looks menu for a function is set up like this:

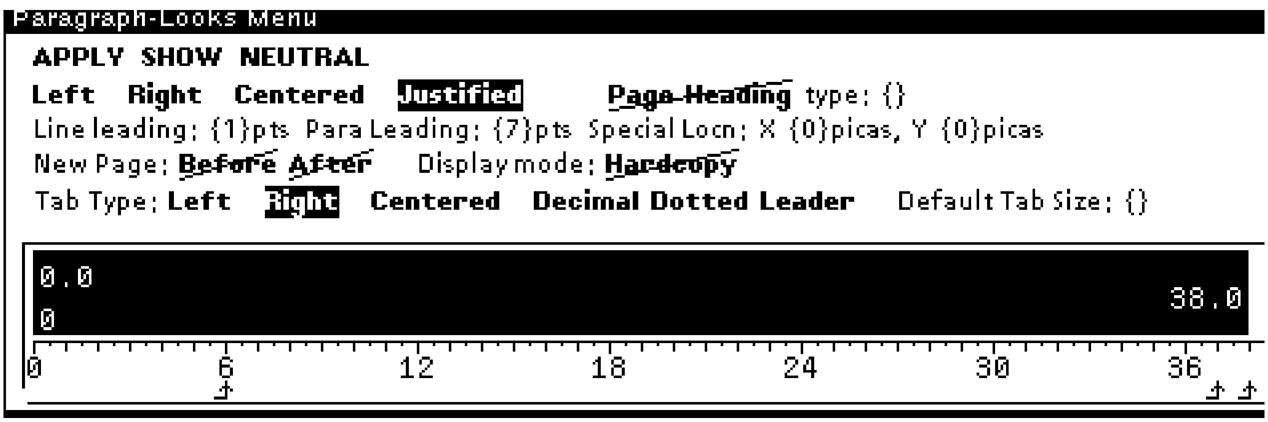


The function name is in 10-point regular Modern, all caps. Arguments are in 10-point italic Modern, in all caps, mixed case, or lowercase, as they appear in the system.

If the function description is more than one line long, the runover arguments should be indented under the function name and the word [Function] placed on the last line of the argument list, like this:

(MAKE-ARRAY *INDICESLIST &KEY :ELEMENT-TYPE:ˆ*
INITIAL-ELEMENT :INITIAL-CONTENTS :ADJUSTABLE:ˆ
FILL-POINTER :DISPLACED-TO :DISPLACED-INDEX-OFFSET) [Function]

The Paragraph Looks menu should be set as follows to produce this type of argument format:



Long function descriptions should have two points leading between lines. To space them correctly, hold down the meta key when typing the carriage returns so that TEdit breaks the lines without creating separate paragraphs.

Parentheses are in 10-point regular Modern type. The type of definition is in 10-point Modern, caps and lowercase, enclosed in square brackets, and flushed right with a right tab set to 38.0. Note that tabs are used to indent the second and third lines of arguments.

Variables look like functions, except that the word "Variable," enclosed in square brackets, follows the variable name.

Examples of code should be in 10-point Terminal font (but not function names, commands, file names, and the like). If 10-point Terminal is not available on your printer, use 8-point Terminal. Code examples should have two points line leading and no paragraph leading.

Quotation Marks, Bullets, and Dashes

We recommend using TEdit's "expanded abbreviations" to produce professional-looking quotation marks, bullets, em-dashes—used to separate text phrases—and en-dashes (used to indicate inclusive numbers, as in "pages 3–6").

· To produce a bullet, type a lowercase b, select it, and type Control-X.

· To produce an em-dash, type a lowercase m, select it, and type Control-X.

· To produce an en-dash, type a lowercase n, select it, and type Control-X.

HEADS

There are four levels of heads in the Lisp Users' documentation: chapter (level 1) heads, level 2, level 3, and level 4 heads.

Note: A head that falls at the bottom of a page (a "widow") is undesirable. You eliminate a widow by selecting it, then applying the Before option of the New Page command in the Paragraph Looks menu.

The Chapter Head

The chapter head appears at the beginning of the document and identifies it. The heading "Lisp Users' Template," above, is a correctly formatted Lisp Users' chapter head.

When submitting a Lisp Users' module on floppy disk format the chapter head as follows:

Module Name: your Lisp Users' Module (all caps, 12-point bold Modern)

Your name and ARPANET address (if you have one) in 10-point Modern

The Level 2 Head

Level 2 heads identify major sections of a document. The level 2 heads for the Lisp Users' documentation are in 10-point bold Modern, all caps.

The Level 3 Head

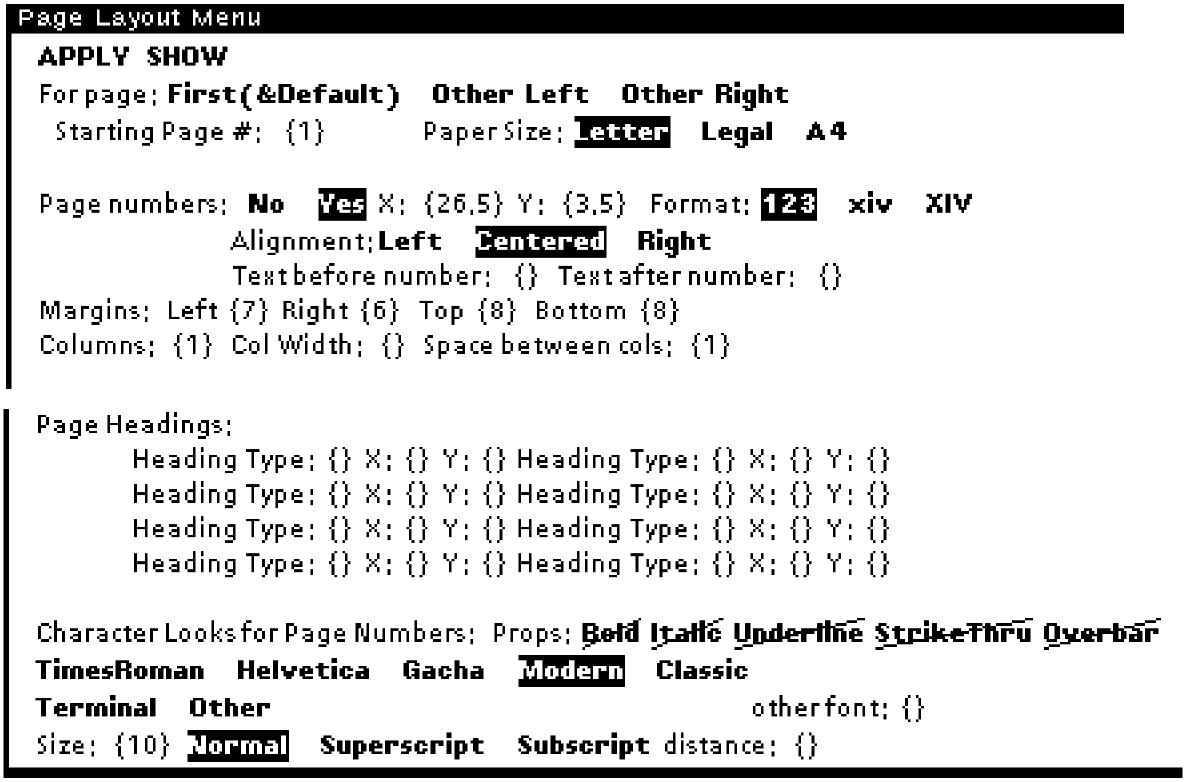
Level 3 heads identify subsections of a document. For the Lisp Users' manual, they are in 10-point bold Modern, caps and lowercase.

The Level 4 Head

Level 4 heads identify the lowest level of subsection in the Lisp Users' documentation. They are in 10-point regular Modern, caps and lowercase, underlined.

PAGE NUMBERS

Page numbers are specified and applied in the Page Layout menu. First, specify the alignment of the page numbers to be centered, with the *X* position being 26.5 picas and the *Y* position 3.5. Then specify the character looks to be 10-point regular Modern. Finally, apply the page numbers to the First(&Default) pages.



After you submit your document, XEROX AIS will add running heads, put in additional formatting, and provide final page numbering to assemble it into the Lisp Users' manual.

MANUAL TEMPLATE

The following page has a sample template for the information required in a Lisp Users' module. Use this template to produce your module documentation.

Module Name:

>>Module Name<<

By: >>Your Name<< (>>Your net address<<)

Uses: >>Other modules necessary to run this one<<

>>Type INTERNAL here if the file is for Internal Use Only<<

This document last edited on >>DATE<<.

INTRODUCTION

>>This paragraph should be replaced by an overview of your module. The information on the previous pages explains the documentation conventions to be used for each Lisp Users' module.<<

MODULE EXPLANATIONS

>>Functions, Variables, and Lisp Code Examples<<

It is usual to first give the name of a function, then describe its purpose and each of its arguments.

Module explanations may have several level headings.

Be sure to include the following information in any module explanations:

• any file dependencies

• definitions of all arguments

• module, function variable, etc. limitations

• a liberal number of examples for all functions, variables, etc.

ACE

By: Michel Denber (Denber.wbst@Xerox.com) Compiled for Medley by Larry Masinter (Masinter.PA@Xerox.COM)

Files: ACE.LCOM

Data files: ACE-APPLEDEMO.ACE, ACE-BOUNCINGBALL.ACE, ACE-FOUETTE.ACE

Animation Compiler and Environment

Introduction

ACE is a system for computer-assisted animation. It is based on the traditional cel-oriented animation process with the computer taking over many of the tedious jobs. You enter a succession of frames which represent a *sequence*. The system then plays back your frames to create the animated effect. It lets you draw pictures, enter text, and edit your work. The animated images you make are displayed on the screen in real-time. The two main parts of ACE system are a frame compiler and an environment. The environment provides the editing tools, frame manipulation, and display capabilities. The compiler operates automatically to produce a compressed-storage representation for frames.

You can also use the graphic editing features in ACE to make individual pictures, whether or not they're intended to be used for animation. Finally, you can use the compiler directly to compress any bitmap image so that it take up less space on your disk.

The majority of the code for ACE was originally written by Paul Turner, a student at the University of Rochester. I am currently maintaining the system. Please send all bug reports, comments, and suggestions directly to me, Denber.WBST, or Denber.WBST@Xerox.COM (Arpanet). This document describes the features available in ACE version 2.1.

Background

In this document: *holding* the mouse on a menu selection means to press down a mouse button on a menu item (inverting the item) and keeping it down for about 1.5 seconds (at this point, you can release the button or move to another selection). *Clicking* the mouse means pressing a mouse button down and releasing it. Unless otherwise stated, the left mouse button is used for selecting items from menus and to click at objects.

In addition to the mouse, ACE supports a graphics tablet (*Summagraphics MM1201*) . [LMM: The graphics tablet hasn't been tested in Medley.] The tablet is more convenient for doing free-hand drawing; in fact, most commercial animation systems include a graphics tablet. The pen has two buttons: the stylus *tip* (which is activated by pressing down on it) and a blue button on the *barrel* of the pen (activated by pressing with the forefinger). We have adopted a convention with regard to the tablet: the stylus button acts like the left button on the mouse and the barrel button acts like the middle button on the mouse.

Terms and System Organization

A *region* is simply a rectangular area.

A *frame* is a region that contains one complete "picture" in an animation sequence; it is a rectangular bitmap with a fixed width and height.

A *sequence* is a collection of frames defining one complete animated segment.

The *current frame* is the frame in a sequence to which operations will be applied. As all the frames in any given sequence are the same size, you may say that one characteristic of a sequence is a region of a particular size. Frames are referred to by number for convenience; the numbering is from 1 to n.

There are two principal windows used in ACE. The *sequence window* is the window on the screen where a particular sequence will be created, edited and displayed. Typically, you define the shape of the sequence window to give you just the area you want to work in, although a sequence can also be edited and displayed in any existing window.

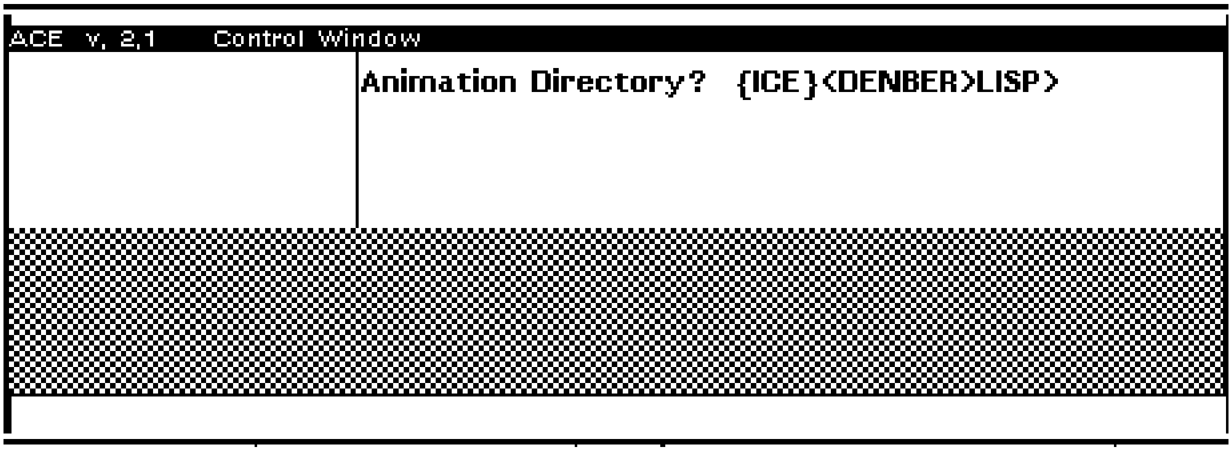
The *ACE Control Window* holds a menu of commands and displays animation state, prompt, and help information. The upper left region in the control window, referred to as the *status region*, tells which frame is currently being displayed (which frame is the current frame); which device (mouse or tablet) is being used in line art and painting operations; what operation is currently being performed; and, the size of a region (width, height) or the location (x, y) of the cursor within the sequence. The upper right portion of the control window is the *prompt region*; it is used to get user input and display helpful information. The bottom part of the control window is a menu of animation functions.

GETTING STARTED

Load ACE.LCOM from your lispusers directory (e.g., (FILESLOAD (SYSLOAD) ACE). When this is complete, type:

    (ACE)

At this point an Ace control window will appear by the cursor. You can place it wherever you wish on the screen. The window initially contains a prompt "Animation Directory?" asking for a default directory to use for storing and retrieving animation files (The default selection is your login directory. Just press the return key to accept the default.) The control window can be moved around just like any other window. While you never need to "quit" from ACE, if you close the control window with the right mouse button menu it permanently aborts ACE; if you then re-type (ACE), the animation system will be restarted from scratch.

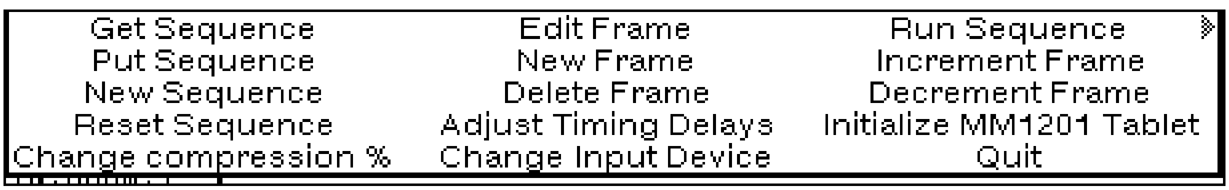


ACE Control Window at start-up

ACE Commands

Main Menu

The menu selections from the control window will now be described; an example of using ACE is given in the next section. The main menu is divided into three columns. The left-most one contains commands that affect the entire sequence, the middle column operates on frames, and the right-most column contains utility commands.



Sequence commands (left column)

Get Sequence Loads a sequence-file from a file server or local disk. You are prompted for the name of the file; incomplete file names will be completed from the directory given as the default ACE directory. You will then be prompted for a window specification. Unless you want to display the sequence in a particular window, select 'create window automatically'.

Put Sequence Saves the current sequence to a file. You will be asked for a file name and given the option to overwrite the existing version of the file (if any) or create a new version.

New Sequence Discards the current sequence (if any), and prompts you for a new sequence by requesting a size for the new sequence. Dragging out a rectangular region with the mouse; the exact size of the region is displayed in the status region of the control window. A blank first frame is created, ready for editing.

Reset Sequence "Rewinds" the current sequence to the beginning (i.e. there is no current frame and the next frame to be displayed will be the first frame).

Change compression % This can be used to change the amount of space compression performed by the animation compiler. For general use, there is no need to ever call this command.

Frame commands (middle column)

Edit Frame Allows editing on the current frame. Brings up a menu of editing options, described in the next section.

New Frame Inserts a new frame after the current frame (ie. before the next frame). The frame editor is then automatically invoked.

Delete Frame Deletes the current frame. The current frame is removed and the previous frame become the current frame. The first frame can not be deleted.

Adjust Timing Delays Lets you set the amount of time (in milliseconds) that any particular frame is displayed; for example, a delay of 50 on the 5th frame would mean that the 5th frame will be visible for 50 milliseconds before the 6th frame is put up. You can change the entire sequence or a single frame at a time. For individual frame setting, you get a menu of frames and their current delays. Holding down a selection will display that particular frame in the sequence window (this is also a convenient way to rapidly move to an arbitrary frame); if you select a frame you will be prompted for a new delay value. When new frames are created, they always get a default delay time of 0.

Change Input Device Lets you select either ’mouse’ or ’tablet’ from a menu. This sets the device to be used for line art and painting operations. The status (upper left) region of the control window always shows which device is active. All menu selections have to be done with the mouse, even when the tablet is being used for drawing

Utility commands (right column)

Run Sequence Runs the *remainder* of the sequence. To run the entire sequence, select Reset Sequence before Run. This command has a submenu with two additional commands:

Loop Runs the entire sequence in a continuous loop. To stop the loop, hold down the space bar. This is checked only at the end of the sequence, so just tapping the space bar may not stop the loop.

Loop part Runs a portion of the sequence in a continuous loop. You can specify the starting and ending frame numbers. To stop the loop, hold down the space bar, as in Loop above.

Increment Frame Displays the next frame and makes it the current frame.

Decrement Frame Goes back to the preceeding frame and makes it the current frame.

Initialize MM1201 Tablet This performs the necessary RS232 port initializing, sets the baud rate, activates the tablet, etc. This must be called after the tablet is plugged in and before it is used. If your tablet doesn’t seem to be responding, it may need to be reinitialized.

Edit Frame Submenu

For the commands described below, it is sometimes useful to know the exact coordinates at which a drawing operation will take place. If you hold the T key down, ACE will put the current coordinates in the status window. Release the key to stop this function.

When you select **Edit Frame,** a sub-menu of editing options appears. Their functions are as follows:

Paint This lets you to paint on and erase bits from the current frame. The painting operation is the standard Interlisp-D window paint command. Either the mouse or tablet can be used (which ever device is currently selected). Pressing the left mouse button or pen stylus draws; the middle mouse or pen barrel button erases. Pressing the left shift key brings up a menu. From this menu, you can quit painting, change the brush size or shape, and the color or texture of the "paint brush". Note: selecting items from this menu requires using the mouse (unfortunately, the tablet cannot be used for menu selecting). For more information on the paint command, please see page 19.20 in the Interlisp manual.

Line Art This lets you add straight lines to a frame by selecting one vertex, dragging out a line, and then selecting another vertex. In this way, an arbitrary string of connected line segments can be created. The left mouse button or pen stylus will "put down" vertices, the middle mouse button or pen barrel stops the line dragging. The right mouse button brings up a menu of line art options (paint or invert drawing; several line width choices); as with all menus, the menu selection must be made with the mouse.

Edit Bits This lets you use the Interlisp-D bitmap editor on the selected frame. The mouse is used to turn specific bits on or off (the tablet is not used as it isn't helpful for this application). A complete description of the the bitmap editor is given in the Interlisp Reference Manual. Always exit the editor by selecting **OK**.

Text Lets you put text into a frame. After selecting the 'Text' option, you will be prompted for font characteristics. Then you point (with the mouse) to where the text should begin and click the left mouse button. You may now type in text from the keyboard and it will show up in the frame. A press of the return key ends text entering (the return is NOT included in the text).

Move Region Lets you move an arbitrary rectangular region in the current frame. You first drag out the region to be moved, then you will be asked what to do with the old image (i.e. leave it alone, erase it). At this point, a ghost image will attached to the cursor and you can position the image in its new location. Clicking the mouse will set the position for the image and then you will be asked how to combine the image (i.e. paint it in, exclusive-or it in).

Combine Region This is similar to move region, except that you can select any region on the screen (not just inside the current frame). After selecting the region, bringing the mouse within the sequence window will show a ghost image; the rest of the procedure is the same as for **Move Region**. This command is extremely useful for bringing images created elsewhere into your frame. For example, you might have a drawing made using Sketch or an AIS file.

Texture Area Fill This lets you fill in an arbitrary closed curve with a texture pattern. You are first prompted to select a bounding region. This reepresents a maximum area beyond which texturing will not occur, in the event that the texture "spills" outside the region being shaded. Next, select a starting point anywher inside the desired region. You will then be offered a menu of predefined textures. You can choose one of these, or create your own by selecting *** Other ***. The area is then filled with that texture. You can then confirm that the right thing happened by clicking left. Click any other button to undo the operation.

Texture Region Fill This is like Texture Area Fill, except that it is used to create filled-in rectangular boxes, rather than arbitrary areas.

Scale Region Lets you change the size of any rectangular area. You first select a region, and then indicate the shape of the scaled area by sweeping it out on the frame. Useful if you know how big you want the result to look but not what percent of the original it is. This command has a submenu:

To a new region Same as the top level Scale Region.

In x and y You first select a region as in **To a new region**, and then indicate the percentage of the original size to scale by, much like selecting reduction or enlargement on a copier. You can set the x and y scale factors independently.

In x only Use this if you only want to scale in the x direction, leaving y unchanged.

In y only Use this if you only want to scale in the y direction, leaving x unchanged.

Clear Region For clearing regions to white quickly. If you select a region within the sequence, it is immediately erased. There is no UNDO for this operation.

Quit - Compile This is the usual way of exiting the editor. This keeps the changes made to the frame and calls the compiler, resulting in adding the frame to the current sequence.

Quit - ABORT Exits the editor but does not update the frame. The sequence will be as it was before you selected **Edit**. Any changes made to this frame are lost.

CONCLUSION

Examples

You might want to see an existing animation before creating one of your own. There are several animation demos included in the Lispusers distribution of ACE: ACE-APPLEDEMO, ACE-BOUNCINGBALL, AND ACE-FOUETTE. The simplest one, BOUNCINGBALL, contains five frames showing a bouncing ball. To see this, start ACE running and select Get Sequence from the main menu. Type the name of the file, including the file server and directory if they are different from your current animation directory. Once the file is loaded ACE will let you position the sequence window. Then select Run Sequence. The system will display the five frames and then stop. To see the ball bounce continuously, select LOOP from the submenu on the Run Sequence command. The ball will now bounce until you hold the space bar down.

The file ACE-APPLEDEMO is a 125 frame sequence which shows an apple getting shot. ACE-FOUETTE is a six frame cycle of a ballet dancer performing a fouette turn.

Animation Hints

Remember that ACE is just a tool - it will not do any animating for you. Our goals were to provide a system that simplified the frame creation process and let you create animation on the computer without having to learn a special animation programming language. However, animation is an art form in itself. Experience is gained only through practice and experimentation.

Avoid moving objects too far between frames or the motion will appear jerky. The D machines screens are designed to mimimize flicker through the use of a long persistence phosphor. Unfortunately, this results in trailing streaks of light behind rapidly moving objects. Sometimes you can use this to artistic effect. It can be reduced by moving dark objects over a light background, rather than the reverse.

Sometimes you can simplify the animation process by creating frames out of order, especially for cyclic animation. For example, the bouncing ball was created by drawing frame 1 and then making a new frame (which is by default a copy of frame 1). Then we backed up to frame 1 and added a new frame between 1 and 2, showing the ball half-way down. Then this frame was copied, yielding four frames. Backing up again and adding the middle frame gave a symmetrical bounce sequence with frames 1, 5 and 2, 4 being identical.

It's often convenient to keep a snapshot of the object you're animating handy in a window next to the sequence window. It can then be brought into the frame whenever needed, for example in the case where it is being modified in some way. You are not limited to editing images with the ACE editor. In particular, you may want to use the *Sketch* editor (an Interlisp-D library package) to modify images, and then bring them into the sequence window for compilation.

References

Denber, Michel, Paul Turner, "A differential compiler for computer animation", to appear in *Computer Graphics*, **20**:3, 1986 (Proc. SIGGRAPH '86)

Fox, David, Mitchell Waite, *Computer Animation Primer*, McGraw-Hill, N.Y. 1984

Magnenat-Thalmann, Nadia, Daniel Thalmann, *Computer Animation: Theory and Practice*, Springer-Verlag, Tokyo, 1985

ACE Maintainer's Notes
Created: 4/29/85
Revisions:

INTRODUCTION
This document is intended for programmers who intend to fix, update or otherwise significantly modify the ACE code. A general description and several specific points about the Animation Compiler and Environment will be brought out This is neither a user's guide nor a guide for animation. A user's guide may be found filed under {ICE}<TURNER>ACE>ACE-USERS-GUIDE; a technical document is filed under {ICE}<WHOKNOWS>ACE>WHOKNOWS.

OVERVIEW
The Animation Compiler and Environment is a program for creating, editing, displaying and storing/retrieving frame-oriented animations. It is based on traditional animation principals; successive frames (each frame usually slightly different from the previous frame) are displayed at a rate fast enough to create the illusion of motion (atleast 10 frames/second; usually more). To the user, the ACE system presents a sequence of whole frames (represented as bitmaps with the same width and height); the analogy being a stack of paper. In reality, however, the program maintains virtual frames. This decreases the storage requirements and allows the very rapid display of frames.

VIRTUAL REPRESENTATION
The virtual frames maintained by ACE are differential frames; that is, a frame only contains the information that is different from the previous frame. By this mechanism, the first frame is a complete frame (complete bitmap) providing the intial information. The second frame, then, is just the differences that exist between the first frame and a hypothetical complete second frame (i.e. just the differences between two bitmaps). And so on with all successive frames. The details of data structures and the compilation process are given below.

TRILLIUM vs STAND-ALONE
Currently, ACE is designed to operate as either a stand-alone program or in the Trillium environment. Conceptually, as a stand-alone, ACE should be considered more procedure oriented. When ACE is called (top level fn: (ACE)), a control window (ACE.CONTROL.WINDOW) is brought up which contains a menu. This window and menu remain up and stay active; once ACE is activated, it becomes a part of the user's current environment. In Trillium, ACE operates more functionally; that is, ACE is called as a one-time function which returns a value (although, ofcourse, it produces side-effects). The control window and menu are only active during the function call and are taken away when the ACE session is concluded. Also, a special animation-run-time function is provided for Trillium to use outside of ACE to run animation item types. At present, Trillium expects all editing (and most loading and storing) of animation sequences to be performed inside ACE.

ORGANIZATION
ACE is organized in a top-down fashion which allows a general to specific description of it. What the components do and how (where appropriate) will now be outlined.

FILES: There are four files to the ACE system (and one utility: RS232): ACE, ACE-MAIN, ACE-EDIT and ACE-PRIM. The divisions exist partly for organizational purposes and partly for ease in locating and working with segments. The ACE file contains some macros and variables, but exists primarily to load in all the necessary files. ACE-MAIN contains all the functions necessary to define, manipulate, run and load/store animations. ACE-EDIT contains all the editing functions (including interfacing to the MM1201 graphics tablet). ACE-PRIM contains the compiler funtions; it is only concerned with compiling two frames to create a virtual frame.

ACE-MAIN
This is the guts of the animation environment. There are four main divisions in MAIN (you can see them by looking at the file coms): Top level fns, Trillium-geared fns, I/O fns, and "helper" fns. In addition, there are several macros and a GLOBALVARS declaration. All the main functions that work with animation sequences are in the first division (including the startup fn: ACE). The I/O section is just for reading and writing files; user input and output fns are in the helper fns section. Control window operation and clipping region fns are also located in the helper fns section.

ACE-EDIT
This segment also consists of four parts (again, note the file coms). The first section provides

entry/interfacing to MAIN and calls to the actual editing routines. LINEART contains all the major fns for line art drawing. The third part contains all the other editing fns (e.g. painting, text, moving, etc.). The last part consists of tablet access fns and helper fns (all MM1201 code is located in this section and the code to read the current input device). The RS232 package is used by ACE-EDIT to read the MM1201 Summagraphics tablet. At present, the RS232 package (RS232.DCOM) is loaded by the ACE file. However, the maintainer should keep aware of changes in this package and its whereabouts.

ACE-PRIM; How the Compiler Operates:
The compilation process is concerned only with compiling two frames at one time (MAIN and EDIT take care of administration). The entry function is ACE.COMPILE.FRAME with arguments: BM.ORIG (the first or original bitmap), BM.CHANGED (the successor bitmap), VERTICAL.BLOCK (defines the height in scanlines for primitive regions of change; the horizontal component is 16 so as to take advantage of the 11xx series word size), and THRESHOLD (the percent as an integer representing the minimum amout of "changed area" allowable in a combination; more on this later). Take note that there are two variables defined in PRIM: ACE.PIXPERWORD (16, just the 11xx word size) and ACE.BITMAP.MASK (a mask used for ignoring extra bits in the last raster word of a bitmap).
The compiler works by comparing each word in BM.ORIG to BM.CHANGED (NOTE: these bitmaps must have the same dimensions) VERTICAL.BLOCK words at a time. If a changed word is found, a region specification is entered on list denoting this "region of change". After the whole area of the bitmaps is checked, the compiler attempts to merge these smallish changed-regions (see ACE.MAX.REGIONS). The algorithm first attempts to combine regions which result in no space wastage (this is always desirable). When no more 100% regions can be formed, the algorithm tests all pairings of primitive regions to find the highest efficiency (i.e. least space waste). If this effeciency is greater than or equal to TRESHOLD, the two regions are combined, otherwise, the algorithm terminates. The region merging process is the slowest aspect of ACE; the problem of merging regions is thought to be NP-complete.

DATA STRUCTURES
- A sequence is of the form: (FRAME FRAME FRAME ...).
- A FRAME is of the form: (DELAY BLITS).
- A DELAY is an integer representing a time delay in milliseconds.
- BLITS is a list: (BLIT BLIT BLIT ...).
- A BLIT is of the form: (BITMAP XCOOR . YCOOR).

A BLIT is essentially a small changed area with information on where it should be placed (relative to the animation). The record defintions for the above structures are in the file ACE. The compiler uses the structures: REGION : (LEFT BOTTOM WIDTH HEIGHT); a modified REGION : (REGION . AREA) for merging; and lists or both of these types.

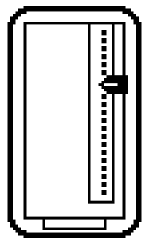
GLOBALVARS
The following are important global variables with explanations where needed.

- ACE.CONTROL.WINDOW contains the top-level menu and status information.
- ACE.DIRECTORY is a default directory used to store/retrieve files.
- ACE.SEQ.WINDOW the current window where animations are displayed.
- ACE.SEQ.WIDTH and ACE.SEQ.HEIGHT refer to the current sequence.
- ACE.SEQ.WINDOW.XOFF and ".".".YOFF the offset in the ACE.SEQ.WINDOW.
- ACE.CURRENT.SEQUENCE points to data which is the current animation sequence.
- ACE.CURRENT.SEQUENCE.NAME for retaining file name information.
- ACE.FRAME.TAIL tail of frames starting one after the current frame.
- ACE.CURRENT.FRAME a tail of frames starting with the current frame.
- ACE.VERTICAL.BLOCK value to use when compiling (see above on compiler).
- ACE.AREA.THRESHOLD for compiling.
- ACE.RUNNING.UNDER.TRILLIUM T if ACE was called by Trillium.
- Various .CURSORs are just cursors.

There are also a GLOBALVARS list of menus in MAIN and EDIT. The approach on menus was that they should be created only once to save both time and space.

ADDRESSBOOK

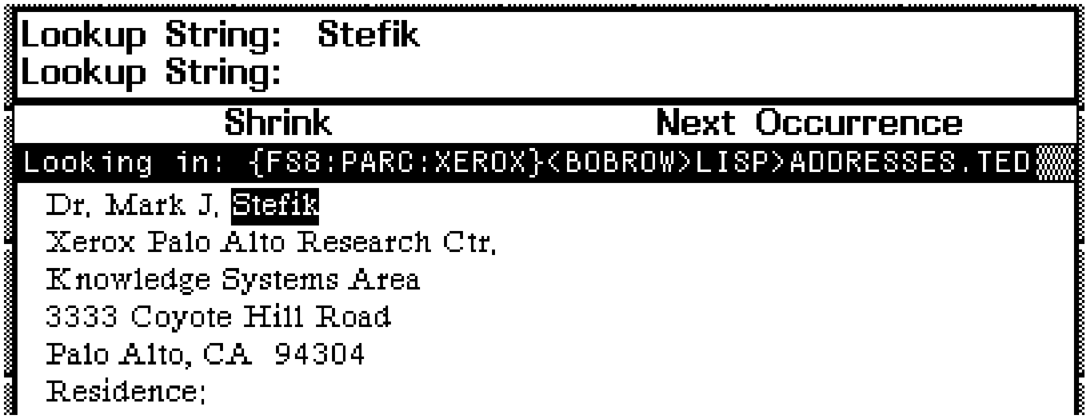
By: dgb (Bobrow.pa@Xerox.com)



INTRODUCTION

The ADDRESSBOOK package provides quick and easy access to on–line address books or phone directories. It allows you to copy (shift select) from entries found in the book, for example, for use as a letter or electronic mail address. When you load the ADDRESSBOOK package, the icon shown above will appear on your screen. Opening this icon will provide a window interface to a simple search process. To find an entry containing any string in one of your *AddressBookFiles*, type the string followed by a return. The ADDRESSBOOK program will quickly search through the files and show you an occurrence of the string typed. The located string is shown in inverse video. The title of the window will contain the name of the file in which the entry was found. You can use a name, part of the address or any keywords to locate the appropriate part of the text. The search ignores case; e.g. "bobrow" matches "Bobrow". The text of the document is scrollable, and any portion can be shift selected into another document.

Type carriage return, ^X, or click on **Next Occurrence** to search further in the files for the same string. If no (further) occurrences are found, the text window will display a message indicating the failure. Searching again after failure will start the search from the beginning of all the files, using the same lookup string. Typing a new string can be repeated as many times as you like.   When you are done, just SHRINK the window back to its icon by using the Shrink selection in the title bar .



Example ADDRESSBOOK window

Required Files

This package automatically loads LOOKUPINFILES.

Variables

AddressBookFiles [Variable]

AddressBookFiles is a list of files that contain entries to be searched. This is usually set in the INIT.LISP file. In the ADDRESSBOOK package it is initially set to PHONELISTFILES, to make it backwards compatible with PHONE-DIRECTORY. The *AddressBookFiles* can be any unformatted or TEDIT formatted files, with any number of lines per entry. A typical value for PARC users (as defined for PHONELISTFILES in PARC-INIT) is

```
({PHYLUM}<REGISTRAR>PARCPHONELIST.TXT
 {INDIGO}<REGISTRAR>ISDPHONELIST.TXT).
```

Address-Book-Pos [Variable]

Address-Book-Pos is the initial POSITION for the ADDRESSBOOK icon. This is defined as an INITVAR in the file, so you can set it before loading the file. The default value is

```
(create POSITION XCOORD ←  970
                 YCOORD ← (DIFFERENCE SCREENHEIGHT 90)).
```

This places the icon in the upper right corner of the screen.

Address-Book-Region [Variable]

Address-Book-Region is the initial REGION for the ADDRESSBOOK window. This is defined as an INITVAR in the file, so you can set it before loading the file. The default value is

```
(CREATEREGION 300 (DIFFERENCE SCREENHEIGHT 500)
               400 200).
```

This places the window in the middle of the screen.

Notes

Starting or Restarting Address Book

Evaluating (MakeAddressBook) will create an address book window and process. This may be useful if you accidentally close the window.

Caching Files

When you first open the ADDRESSBOOK window, the program will copy the *AddressBookFiles* to {CORE}, significantly speeding up queries. Bugging in the title of the ADDRESSBOOK window with the left or middle mouse button will produce a menu with an option to recache the files on *AddressBookFiles*.

Editing Your Files

To edit the file in which an entry is found, click middle button in the title of the ADDRESSBOOK window, and select the option "Edit file named in window title". A TEDIT process editing the file will be set up. This process is independent of the lookup process. To select the file to be edited, rolloff the above item, and select "Select file to edit". A menu of files used by the Lookup process will be presented to you. Selecting one will cause that file to be edited.

To make editing changes visible to the lookup process, PUT the file in TEDIT; when it is done, recache the the file in core. To recache just the file edited, (the one specified in the title bar of the window), select the option "Recache file named in window title" in the middle button title bar menu. You can recache all files by selecting the option "Recache all files" in the title menu (a subselection of the item "Recache file named in window title".

Adding to the List of Files

To add to the list of files being used for lookup, select the option "Add new file" in the title bar menu. This file will be added, and cached in core.

Deleting a file from the List of Files

To delete from the list of files being used for lookup, select the option "Delete file from list" in the title bar menu. This file will be deleted from the list of files to be searched.

AIREGIONS

(Active Irregular Regions)

By: Greg Wexler (Wexler.pasa@Xerox)

and

By: Jim Wogulis (Wogulis@ICS.UCI.EDU)

New Owner: James Turner (Turner.Lexington@Xerox.com)

Uses: FILLREGION, AIREGIONS-DEMO

INTRODUCTION

The purpose of this package is to provide menu-like operations on irregularly shaped regions within a window and make available general functions that allow users to create their own applications using irregularly shaped active regions. An added feature of AIRegions is that multiple IREGIONs may be activated by selecting the intersecting area of those IREGIONs. (Throughout this document an irregularly shaped region will be referred to as an IREGION).

DESCRIPTION

Virtually all of the features of menu selection have been implemented in this package: ease of menu creation, item-selected shading, quick response to selection, and execution of an associated function. Yet, this package adds one additional feature without any degradation to the quality and efficiency of menu implementation: the selection of any irregularly shaped region from any point within that region, and without any unsightly cosmetic change.

In describing the package by means of an example, picture a map of the world, or better yet, of a particular country broken up into its individual states and/or provinces. Suffice it to say that these regions are not square but irregular in shape and that they are bordered by solid lines, as they are on a common map. Unlike the menu package or ACTIVEREGIONS package, AIRegions allows you to select any of these pre-set states/provinces just as if your are making a menu selection of an item. One of the nice aspects of this package lies in the fact that the package does NOT make any costmetic changes to the irregularly shaped region, like providing some small box within the region to button in. Simply button your mouse within the solidly bordered region, anywhere in the region, and it will shade it to your particular shade and execute your defined function.

Functionality provided:

The functions in this package allow the user to work with familiar concepts: creating and implementing windows and menus. The examples provided within this documentation should be sufficient for the user to begin setting up irregularly shaped regions.

(CREATEIR *window shade buttoneventfn helpstring region poslist*) [Function]

window: the window which will contain the irregular region.

shade: can be either a number between 0 and 65535 for a 4 by 4 shading or a 16 by 16 bitmap (if *shade* is NIL then the default is black, 65535).

buttoneventfn: the function called when the region is selected. The arguments that are passed to the function are: the window containing the IREGION, the IREGION record itself, and the button which selected the IREGION.

helpstring: the string that is placed in the PROMPTWINDOW when the mouse is held over the item for a few seconds.

region: if specified, will be the region relative to *window* in which the IREGION can be found. (If *region* is NIL, the user will be prompted to sweep out a region within *window*.)

poslist: If specified, will be either a position or list of positions relative to *window* that are the starting points for the FILLREGION routine (i.e. a point within the desired IREGION). (if *poslist* is NIL, the user will be prompted for a position until he/she selects outside of *region*.)

Description of use: This is the first function that is called when actually setting up an irregularly shaped region to become sensitive to button activity. If the *region* argument is not set, then the cursor changes its shape and prompts for a region to completely surround the IREGION within the desired active window.(Note: That it is best to surround the desired IREGION as close as possible since this will save on execution time and memory useage.) A thin box will appear temporarily where the IREGION was scanned. If *poslist* is NIL, then the cursor changes into a TARGET symbol. The user should left-button mouse within desired active IREGION. Note: the IREGION must be surrounded by a border that FILLREGION can use to define the active area. Any gaps in the IREGION will cause the next routine to fill the region and anything outside with the shade provided. Mistakes can be corrected by using the REMOVE.IREGION function described below and PAINTing in the gap to retry. After left-buttoning within the desired active IREGION, the cursor continues to remain in its TARGET state. If the IREGION is split up into many different parts, those parts may be selected with the left-button also making them all active concurrently. However, when one is finished activating that one IREGION, then she/he should left-button outside of *region*. This function must be called for each desired IREGION.

Examples:

```
(CREATEIR  window  21930  'myfunction  "This is the helpstring")

(CREATEIR (WHICHW) 1234 'MY.SELECTED.FN "This is the helpstring"
'(0 0 20 30) '((12 . 15)(2 . 29))
```

(SURROUNDIR *window shade buttoneventfn helpstring poslist inside.pos*) [Function]

window: the window which will contain the irregular region.

shade: can be either a number between 0 and 65535 for a 4 by 4 shading or a 16 by 16 bitmap (if *shade* is NIL then the default is black, 65535).

buttoneventfn: the function called when the region is selected. The arguments that are passed to the function are: the window containing the IREGION, the IREGION record itself, and the button which selected the IREGION.

helpstring: the string that is placed in the PROMPTWINDOW when the mouse is held over the item for a few seconds.

poslist: If specified, a list of positions relative to *window* that are the edge points for the FILLREGION routine. If NIL, the user will be prompted to define the outer border of the region desired to be active. Holding the SHIFT key will define the last point used in defining the edge. If this field is non-nil, *Inside.pos* must be specified.

Inside.pos: If specified, this would be the inside position in which the Fillregion routine would begin filling from. If *poslist* is non-nil, then this field must be specified.

Description of use: Like the CREATEIR function, this function creates IREGIONS. However, the functionality of this routine is quite different. There are times when you do not care what is within a particular region. Say, for example, you have a map of some country and you wish to surround a particular region of the country with an IREGION as you wish to denote an area rich in some mineral deposit or some other characteristic. Such a characteristic is oblivious of the borders of the country's states or provinces, streams, rivers,etc., yet you would like to make active a very general area. Upon calling this function, you are prompted to button around the area of interest. And so, in viewing the crosshairs cursor, you begin buttoning about specifying the border of the area you wish to make active, independant of what is inside it. To stop being prompted for the next edge, simply hold the SHIFT key on the keyboard, (either one will do), as you make your last button selection. At this point, the lisp DRAWCURVE function will take effect and draw the closed region you've defined. Note that the first and last points do not have to touch as the DRAWCURVE routine will connect them for you. You will also be prompted to button within the region you've marked. It is here that the Fillregion routine will begin filling your region from. When complete, this function adds the IREGION to the window and returns the iregion added.

Examples:

```
(SURROUNDIR  window  21930  'myfunction  "This is the helpstring")

(SURROUNDIR (WHICHW) 1234 'MY.SELECTED.FN "This is the helpstring"
'((5 . 5)(6 . 50)(50 . 50)(50 . 7)) '(10 . 10)))
```

(ADD.IREGION *window iregion*) [Function]

window: the window to which the iregion is to be added.

iregion: the IREGION to be added to window.

Description: This function will add iregion to window which will then allow mouse selection of that IREGION.

(REMOVE.IREGION *window iregion*) [Function]

window: the window in which the *iregion* exists.

iregion: the IREGION you wish to remove from *window*.

Description: This function removes the region from a list of active irregular regions which is stored as a window property of the window. The list of irregular active regions can be found by evaluating: (ALL.IREGIONS *window*)).

(INTERSECTING.IREGIONS? *window flg*) [Function]

window: a window.

flg: either T or NIL

Description: This function sets up window to allow selection of intersecting iregions. If two or more iregions overlap and this function had been called with flg = T, then when the overlapping region is selected, all of those iregions will be high-lighted and each IREGIONs BUTTONEVENTFN will be called. If flg is set to NIL, then the last IREGION created in that intersection of iregions will be selected. (Please be aware that intersecting iregions *might generate effects that you do not wish to have*. That is, if you leave the iregion "ON" (the exact same thing you see when you hold the mouse button down on the iregion, done by inverting that iregion) and create another iregion intersecting with the first, then the mask of the second would have a partial image of the first. At this point, buttoning in an area where both regions interesect might show everything but the intersection of those regions. Sometimes, it all depends on the *order* that they are created and what iregion's mask is left on or off. Shades that are "negatives" or "equals" of one another might make matters more complex than necessary when they are intersected. It is recommend that you play with this function in order to understand how it actually works so that when you work it into your application you'll have a better idea of the functionality and end-results). If this becomes a problem, an EDIT.MASK function has been provided so that you may edit the mask of the iregion by hand. Currently, there are no programmatic methods for doing this.

(ALL.IREGIONS *window*) (Function)

window: a window containing IREGIONS.

Description: This function returns a list of all the IREGIONS attached to *window*.

(DOSELECTED.IREGION *window iregion button*) (Function)

window: the window associated with iregion.

iregion: the iregion to be activated

button: the button which selected iregion.

Description: Applied iregions BUTTONEVENTFN to window, iregion and button. This provides a programmatic way of activating a given IREGION. This does not invert the iregion.

(EDIT.MASK *iregion*) (Function)

iregion: the IREGION whose mask you want to edit.

Description: This function is provided for buttoning in places where the MASK is not set. More explicitly, TARGETing a region (while creating the regions) specifies the places where the FILLREGION routine is to create a mask. For example, if a US state contains many rivers one pixel wide, the FILLREGION routine will fill around the river, but not the river itself. This means that when the mouse is positioned on the river, the region will not shade because the mask does not have that bit turned on. However, if the mask is edited and the rivers filled in, buttoning on those rivers will activate the IREGION.

(INVERT.IREGION *window iregion*) (Function)

window: the window in which the *iregion* exists.

iregion: the IREGION targeted for shading.

Description: This will highlight the *iregion* with that *iregion*s shade. Calling it a second time will low-light it.

(IREGIONP *iregion*) (Function)

iregion: the IREGION to be tested.

Description: This function returns NIL if *iregion* is not an IREGION datatype and returns *iregion* if it is an IREGION.

(IREGIONPROP iregion prop newvalue) (Function)

iregion: the region of which you are setting/requesting the property.

prop: the property in which you are interested.

newvalue: the new value to be assigned to *prop*.

Description: As with WINDOWPROP, if newvalue is not specified, it will return the current value of the *iregion's* property. If newvalue is specified, then the property will be reassigned with that value. If a prop name is not one of the fields of an IREGION record, it will be stored in property-list format on the USERDATA field of the *iregion* record.

IREGION fields:

BUTTONEVENTFN - function called when *iregion* is selected.
USERDATA - property list format for user properties (similar to WINDOWPROP).
REGION - region relative to the window that surrounds the *iregion*.
MASK - a bitmap the same size of REGION that is blackened where the *iregion* is active.
SHADE - the shade number or bitmap used to shade the region.
HELPSTRING - the string that is printed in the *PROMPTWINDOW* when a region is held.

Examples:

```
(IREGIONPROP  iregion  'SHADE)      -- returns shade of iregion
(IREGIONPROP  iregion  'SHADE  21930) - assigns new shade to iregion.
```

(SHOW.ALL.IREGIONS *window shade delay*) (Function)

window: the window in which the IREGIONs exist.

shade: the shade with which the iregions will be shown.

delay: the time (in milliseconds) between which each IREGION is displayed . (if *delay* is NIL, then a default of 500 is used.)

Description: This function will shade and unshade in *shade* (black is used if *shade* is NIL), each IREGION that has been created in the particular window. This is especially useful when the user has lost track of the number of IREGIONS within a window.

(WHICH.IREGIONS *window posorx y*) (Function)

window: the window in which the IREGIONs lie. (if *window* is NIL, default is window to which mouse points).

posorx, y: the location within the window where the IREGIONs can be found. These points must be local to the window's coordinates...not the screen. (if *posorx* is a position, then it will be used, otherwise if x or y are not numbers then the current mouse position is used.)

Description: Will return either NIL or the list of IREGIONs found in *window* and specified by *posorx, y*.

Examples:

```
(WHICH.IREGIONS)
(WHICH.IREGIONS  MY.WINDOW  50  23)
(WHICH.IREGIONS  MY.WINDOW  '(50 . 23))
```

Saving IRegions

IREGIONS can be saved on a file by setting a variable to be the value returned by ALL.IREGIONS. This variable can be saved by using the file package command, UGLYVARS.

Example:

```
(SETQ IRS (ALL.IREGIONS (WHICHW)))
(SETQ SAVEIRSCOMS '((UGLYVARS IRS)))
(MAKEFILE 'SAVEIRS)
```

The file SAVEIRS can be loaded and IRS will be set. You can then add IRS to a window by doing:

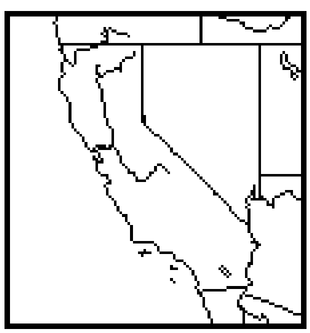
```
(WINDOWPROP (WHICHW) 'IREGIONSLIST IRS)
(WINDOWPROP (WHICHW) 'BUTTONEVENTFN 'IN.CURSOR.REGION)
```

Caution: Some properties on the USERDATA field of an IREGION might not be saved correctly such as a window which can not be saved on a file.

Window images can be saved on a file by creating a bitmap the same size as the window, BITBLT from the window to the bitmap, and then saving the bitmap with the file package command VARS.

Example use of the AIRegions package:

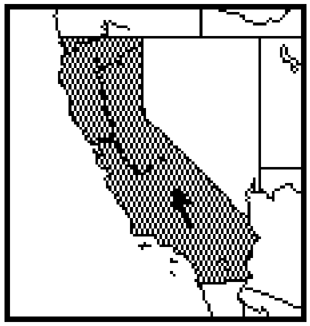
1. Open a window...about 1/4 of a screen.
2. Use the paint function provided when you right-button in the window and paint a picture.



3. With your mouse in this painted window, type in:
   `(CREATEIR (WHICHW) 21930)`
4. The cursor changes shape and prompts for creating a region similar to the prompt for creating a window. In this case, span a region that contains California.
5. When you are done, and the mouse button is released, the region spanned will remain temporarily on the screen. The cursor changes into a target and now prompts for a left-button within the region.

Select somewhere in California. When done, left-button the mouse outside and away from the temporarily blocked off region. (If you want to continue selecting areas of the same irregular region, in this example, the upper left corner of California, then button that area within the squared off region. As you can see, your irregular region does not necessarily have to be connected).

6. To test it out, simply button anywhere in California and it will fill to a nice shade of grey, as we have just set it up to do:



7. To create more active irregularly shaped regions, follow steps 3 through 5 above. If you want to set the selection of one of the regions to activate the execution of some function that calls RINGBELLS, and have the region shade to black upon selection, type in the following in the top level typescript window keeping the mouse within the painted window.

```
(CREATEIR (WHICHW) 65535 'IR.TESTFN)

(DEFINEQ (IR.TESTFN (LAMBDA (WINDOW IREGION BUTTON)
      (If (EQ (QUOTE LEFT) BUTTON)
        then (RINGBELLS 2)))))
```

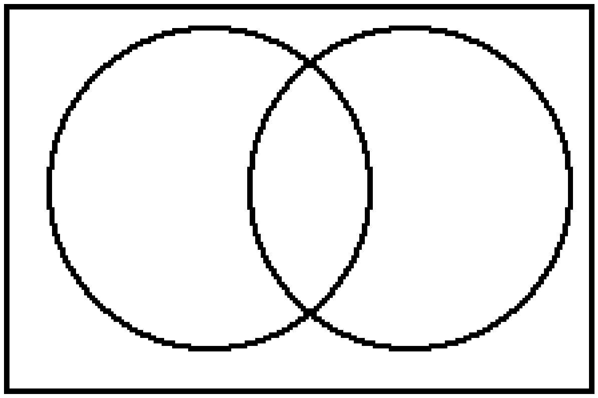
Span the cursor out over another state/region and repeat steps 3-5 above. When you button in this IREGION, the IREGION will temporarily shade black, and call the RINGBELLS function. Note that like menu selection, the function is called only when you release the button within the region. If the mouse button is held down and you move over the created IREGIONs, they will shade and unshade as you enter and exit them.

Note: if you wish to create your own shades but don't know what shades correspond to which numbers, call the function (EDITSHADE) and begin selecting points that you want shaded. When you are done, the function will return the appropriate shade number. You can also use 16x16 bitmaps for the shade of an IREGION (try (EDITBM (BITMAPCREATE 16 16)))

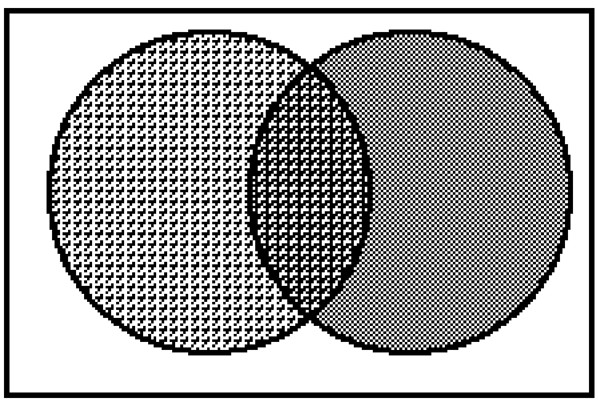
DEMO PACKAGE: To run the demo package, load AIRegions-Demo.

Intersecting Iregions

1. Create a window and paint in the following:



2. Now call CREATEIR passing in this window and a shade of 4747 and surround the left circle and select inside that circle and also in the intersecting area for the area fill. Repeat this for the right circle but use a different shade (say 42405).
3. Now, with your mouse in the window, call the function (INTERSECTING.IREGIONS? (WHICHW) T). When you button in the intersection of the two circles, you should get:



4. When the mouse is released inside of the intersecting region, both IREGIONs BUTTONEVENTFN will be called.

Comments and suggestions are welcome.

AISBLT

By: Nick Briggs (Briggs.pa@xerox.com)

This document last edited on September 21, 1988.

INTRODUCTION

The AISBLT module provides a fast(er) interface for reading AIS format files into Lisp bitmaps. It does not provide all the arcane features found in the READAIS module.

CLIENT INTERFACE

The functions provided by the AISBLT module which are intended to be used by clients are

(AISBLT.BITMAP *FILE SOURCE-LEFT SOURCE-BOTTOM DESTINATION DESTINATION-LEFT DESTINATION-BOTTOM WIDTH HEIGHT HOW FILTER*) [Function]

The *SOURCE-LEFT*, *SOURCE-BOTTOM*, *DESTINATION*, *DESTINATION-LEFT*, *DESTINATION-BOTTOM*, *WIDTH*, and *HEIGHT* arguments are interpreted in the same way as the corresponding arguments to BITBLT. *FILE* is either an open stream, or a filename. If a filename is provided it will be passed to FINDFILE, which searches the directories specified by the special variable

AISDIRECTORIES [Variable]

which should be a list of directories where the AIS file is likely to be found.

The argument *HOW* should be one of the atoms FSA, :FSA, TRUNCATE, :TRUNCATE. *HOW* is only applicable in the cases where the source and destination are a different number of bits per pixel (source bpp > destination bpp). If *HOW* is not specified, it defaults to FSA. FSA indicates that the source should be reduced to the bits per pixel of the destination by applying the Floyd-Steinberg dithering algorithm, as described in Newman & Sproull, Principles of Interactive Computer Graphics, pg. 226. TRUNCATE indicates that only the high order bit(s) of the source should be used.

The function

(AISFILEHEADER *STREAM*) [Function]

Can be used to determine whether a file has a well formed AIS header, and what the attributes indicated in the header are. The result of the function is a property list describing the AIS attributes:

:RASTER [Key]

The :RASTER property will always be present. The value is also a property list

:SCAN-COUNT [Key]

An integer value indicating the number of scan lines in the image

:SCAN-LENGTH [Key]

An integer value indicating the number of pixels in a scan line of the image

:SCAN-DIRECTION [Key]

An integer, indicating the direction of the scan. Scan direction 3 is top to bottom, left to right, and is the only scan direction that this package will deal with at this time.

:SAMPLES-PER-PIXEL [Key]

An integer, indicating the number of samples per pixel. This package will only deal with files having one sample per pixel at this time.

:CODING-TYPE [Key]

An unsigned integer indicating the coding type of the raster image. A value of 1 indicates uncompressed array format, and is the only type recognized by this package at this time. For convenience, the constant

AIS-RASTER-CODING-UCA [Constant]

is bound to the value 1. If the raster coding types are extended, more constants will be defined.

The rest of the properties are coding type dependent. For the AIS-RASTER-CODING-UCA file, the following properties are present:

:BITS-PER-SAMPLE [Key]

An unsigned integer, indicating number of bits per sample

:WORDS-PER-SCAN-LINE [Key]

An unsigned integer, indicating how many 16 bit words form a single scan line of the image.

:SCAN-LINES-PER-BLOCK [Key]

A signed integer, indicating how many scan lines are present before there is block padding. A value of -1 indicates no blocking.

:PADDING-PER-BLOCK [Key]

A signed integer, indicating how many padding words per block. A value of -1 indicates no blocking.

:PLACEMENT [Key]

The placement property is optional. The value is a property list with keys :LEFT, :BOTTOM, :WIDTH, and :HEIGHT. The values are unsigned integers.

:PHOTOMETRY [Key]

The photomety property is optional. The value is a property list with keys :SIGNAL (integer), :SENSE (integer), :SCALE (integer), :SCALE-A (pair of integers), :SCALE-B (pair of integers), :SCALE-C (pair of integers), :SPOT-TYPE (integer), :SPOT-WIDTH (integer), :SPOT-LENGTH (integer), :SAMPLE-MIN (integer), and :SAMPLE-MAX (integer).

A complete description of the meaning of the photometry parameters can be found on page 38 of the AIS format description, filed on {indigo}<altodocs>aismanual.press.

Analyzer

By: Maxwell (Maxwell.pa@Xerox)

INTRODUCTION

The Analyzer package is used by the Proofreader (see PROOFREADER). It defines a class of analyzers, of which the proofreader is but one. Later, analyzers will be developed for languages other than English.

AUTOSAMEDIR

By: Mitchell L Model
473 Edgell Road
Framingham, MA 01701

Uses: SAMEDIR.LCOM

INTRODUCTION

This package is an extension to the SAMEDIR Library package so that if AUTOSAMEDIRFLG is non-NIL, MAKEFILE will automatically switch to the directory its file argument was originally made to instead of invoking the SAMEDIR dialogue. (It uses the FILEDATES property of the file.)

Notice and Acknowledgement: This package was developed while the author was an employee of Applied Expert Systems, Inc. (Apex), Cambridge, MA. The author thanks the company for its support and assumes full responsibility for the contents and maintenance of this package.

SOFTWARE REQUIRED

AUTOSAMEDIR.LCOM

SAMEDIR.LCOM

FIXES

Extension to SAMEDIR package so that if AUTOSAMEDIRFLG is non-NIL, MAKEFILE will automatically switch to the directory its file argument was originally made to instead of invoking the SAMEDIR dialogue. (It uses the FILEDATES property of the file.)

AUXMENU

By: David Newman (Newman.pasa @ Xerox.COM.ARPA)

AUXMENU is a Lispusers package that creates a middle-button background menu. This menu acts like the right-button background menu that exists in any Interlisp sysout in most respects. The menu includes commonly used Interlisp functions that require no arguments to be useful.

Global Variables

MiddleButtonBackgroundMenuCommands [Variable]

DefaultMiddleButtonBackgroundMenuCommands [Variable]

MiddleButtonBackgroundMenu [Variable]

MiddleButtonBackgroundMenuCommands is a list of MENU items (in the same format described in the Interlisp Reference Manual) which is used to create the middle button background menu. DefaultMiddleButtonBackgroundMenuCommands is the default value of MiddleButtonBack-groundMenuCommands. Individual users may reset MiddleButtonBackgroundMenuCommands, or they may add to or change it. If the user changes this variable, MiddleButtonBackgroundMenu should be set to NIL. All variables are initialized by the package as it is loaded. All necessary interaction is performed in the promptwindow, and the result of any menu item is printed to the promptwindow if it is non-nil.

Default Menu Items

Login - This item performs a (LOGIN) via the promptwindow.

Greet - This item performs a (GREET).

Logout - This item does a (LOGOUT).

Cleanup - This item performs a (CLEANUP) via the promptwindow.

Reclaim - This item executes (RPT 5 (QUOTE (RECLAIM))).

Closeall - This item closes all currently open files by performing (CLOSEALL).

Open Files - This item lists the currently open files to the promptwindow using (OPENP).

Connect - This item prints the currently connected directory to the promptwindow. The subitems of this item make the connected directory the one shown as the item. The ’Default’ subitem connects the system to the value of the variable LOGINHOST/DIR. The ’Other’ subitem prompts the user for a directory name, makes that the connected directory, and adds it to the menu.

VMem Size - This item prints the current size of the virtual memory file (in pages) to the promptwindow. It uses (VMEMSIZE).

Free Pages - The number of free pages in device DSK are printed to the promptwindow. (DISKFREEPAGES) is the function that provides this information.

Disk Partition - This item prints the name of the current partition to the promptwindow. The printed value is the result of calling (DISKPARTITION).

Volume Display - Turns the volume display window on via (VOLUMEDISPLAY ’ON).

Default Printers - Types the value of DEFAULTPRINTINGHOST to the promptwindow.

File Changes - Lists the result of (FILEPKGCHANGES) to the promptwindow.

BACKGROUNDIMAGES

By: Burwell (Burwell.pa@Xerox.com)

Accessory files:

Background-DurerCat.bitmap
Background-Parc.press
Background-Rhine.press
Background-Steinheim.press
Background-TwoDollar.press
BackgroundMenu.dfasl
BitMapFns.lcom

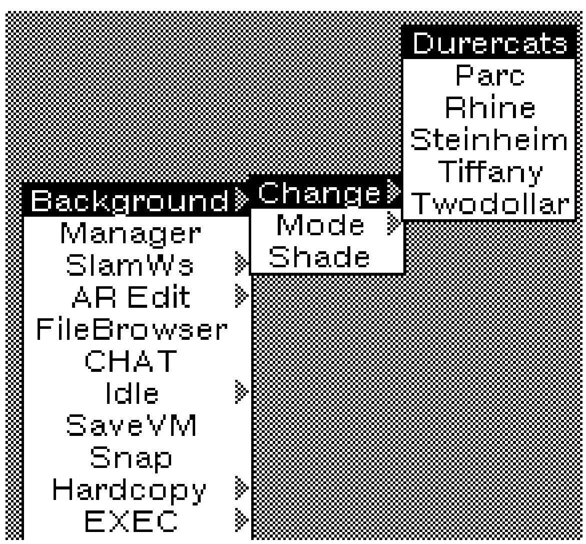
This document last edited on September 8, 1988.

INTRODUCTION

BackgroundImages is a module which makes it easy to apply graphically interesting static images to the background of one's Lisp screen. To use the package in the simplest way, load it and call

(BACKGROUND.SETUP) [Function]

This will put an entry called "Background" on your background menu (in a manner compatible with the module BACKGROUNDMENU), so that it will look something like this:



If (as shown) you select one of the subitems of "Background>Change, " your background will be painted with the image whose name you selected. The background images currently available are

DurerCats: a reflected picture of a cat from an engraving by Albrecht Durer
Parc: a picture of the Xerox Palo Alto Research Center

Rhine: a picture of a village on the Rhine river
Steinheim: a picture of a relatively unfortified castle
TwoDollar: a picture of part of a two dollar bill

If the image you select is a different size than your screen, you may want to control how the image is applied. There are three different image painting modes: "Center," which centers the image on the screen and paints gray in the remaining space; "Tile," which tiles the screen with the image; and "Reflect," which tiles the screen with edge-matched reflections of the image. (This last mode is particularly effective with the DurerCats image.) To change the mode, select one of the subitems of "Background>Mode." To the currently set mode, just select "Background>Mode" itself. Note that once you have changed the mode, to see its effect, you must reapply the background image (by selecting "Background>Change>*ImageName*").

If you want a less busy background, you can use a plain gray. To apply it, just select "Background." To change the shade of gray, select "Background>Shade." Again, to see the effect of the shade change, you must reapply the background shade (by selecting "Background").

DETAILS

Background images to be used with this module must be represented in files that can be read by either HREAD or READPRESS. For convenience, they should be named according to the conventions mentioned below under BACKGROUND.FILES.

BackgroundImages does take some pains to reduce user wait time. First, it is very lazy about file interactions, and defers them until it is quite clear they cannot be avoided. And second, when one selects a background, it is cached so that changing back to it will be significantly faster than fetching it the first time. Since the cached background bitmaps consume quite a bit of space, they can be removed by the GAINSPACE mechanism.

The public interface to this package, more fully described, is as follows:

(BACKGROUND.SETUP *NAMES*) [Function]

Puts an entry on the background menu which enables users to change backgrounds easily. The entry will be labeled "Backgrounds" and if invoked will turn the screen background gray. The entry will have several subitems, each labeled with the name of the background image it will, if selected, put on the screen. The argument NAMES is meant to specify the names of the images; it must be a list either of dotted pairs (whose CAR is the name of an image and whose CDR is the name of the file in which a representation of that image can be found) or of atoms (each of which is the name of an image). If NAMES is NIL, BACKGROUND.SETUP will call (BACKGROUND.FILES) to generate a set of background image names.

(BACKGROUND.FILES *WHICH*) [Function]

Returns a list of dotted pairs whose CAR is the name of an image and whose CDR is the name of the file in which a representation of that image can be found. Generates this list by looking on LISPUSERSDIRECTORIES for files of the form "background-*.bitmap" or "background-*.press"; all such files are taken to be representations of background images. Image representation files that are not named and located according to this convention will have to be specified directly to BACKGROUND.SETUP. If WHICH is T, it will search all the LISPUSERSDIRECTORIES; otherwise it will search till it finds the first directory with background images in it.

(BACKGROUND.FETCH *NAME FILENAME MODE*) [Function]

Causes the image whose name is NAME, and for which there is a representation in file FILENAME, to be applied to the screen background. It is this function which the background menu subitems call to apply new images. If FILENAME is not specified, BACKGROUND.FETCH will attempt to find an image representation file whose name is either "background-*NAME*.bitmap" or "background-*NAME*.press" on any of the LISPUSERSDIRECTORIES. MODE specifies how the image will be applied to the background if it is a different size than the screen. MODE should be one of the atoms CENTER, TILE, or REFLECT; it defaults to CENTER. CENTER causes the image to be centered with a white border around it; TILE causes the image to tile the screen; and REFLECT causes the image to tile the screen such that each tile is a reflection of those adjacent to it.

(BACKGROUND.MODE *MODE*) [Function]

Sets and accesses the mode (as described above) which will be passed to BACKGROUND.FETCH when the latter is invoked from the background menu subitems. MODE, if provided, gives the new mode setting. Returns the previous mode setting.

(BACKGROUND.SHADE *NEWSHADE*) [Function]

Changes the default background shade.

(BACKGROUND.CENTER *BITMAP*) [Function]

Returns a screen-sized bitmap with BITMAP centered in it with a border colored with the default background shade.

(BACKGROUND.TILE *BITMAP*) [Function]

Returns a screen-sized bitmap that is tiled with BITMAP with one of the tiles centered.

(BACKGROUND.REFECT *BITMAP*) [Function]

Returns a screen-sized tiled bitmap such that each tile is a reflection of those adjacent to it and such that the center tile is a copy of BITMAP.

BACKGROUNDMENU

By: Mike Dixon
New Owner: Burwell (Burwell.pa@Xerox.com)

INTRODUCTION

If you love to load all those fun LispUsers packages but can't deal with background menus that look like this:

don't despair! With just a few quick calls you can have a background menu that looks like this:

(don't worry, they didn't disappear, they're just hiding under "Exec").

DESCRIPTION

BackgroundMenu defines several functions for rearranging your background menu to suit your taste.

(BkgMenu.rename.item *item newname*) [Function]

changes the name of a background menu entry

(BkgMenu.move.item *item superitem atend*) [Function]

makes *item* a subitem of *superitem*. If *atend* it is placed after any subitems of *superitem*; otherwise it is placed before them. If *superitem* is NIL *item* is placed at the top level of the menu.

(BkgMenu.reorder *items superitem atend*) [Function]

just like BkgMenu.move.item but moves a list of items. Useful for changing the order of the items in a menu.

(BkgMenu.remove.item *item*) [Function]

throws *item* out of your background menu.

(BkgMenu.fixup) [Function]

BackgroundMenuTopLevelItems [Variable]

BackgroundMenuFixupMode [Variable]

each top level item which isn't on the global BackgroundMenuTopLevelItems is made a subitem of BackgroundMenuSuperItem. If BackgroundMenuFixupMode is 'top they're added before any subitems of BackgroundMenuSuperItem, if it's 'bottom they're added after, and if it's NIL items moved from the top are added at the top and items moved from the bottom are added to the bottom.

(BkgMenu.subitems *item*) [Function]

returns a list of the subitems of *item* (or the top level items, if *item* is NIL).

(BkgMenu.add.item *item superitem atend*) [Function]

adds a new menu item *item* as a subitem of *superitem*. If *atend* it is placed after any subitems of *superitem*; otherwise it is placed before them. If *superitem* is NIL *item* is placed at the top level of the menu

EXAMPLES

As an example of using BackgroundMenu, this is what I've got in my init file (which produces the changes shown above) (note that i've already loaded LISTEN):

```
(BkgMenu.rename.item "Lisp Listener" " Exec ")
```

(* "Lisp Listener" is just too long. the blanks before and after Exec are just there to improve the spacing)

```
(SETQ BackgroundMenuTopLevelItems '(Idle Snap " Exec " Chat PSW TEdit))

(SETQ BackgroundMenuSuperItem " Exec ")

(BkgMenu.fixup)
```

(* Push everything i don't use regularly under the now-renamed Lisp Listener)

```
(BkgMenu.reorder.items BackgroundMenuTopLevelItems)
```

(* and put the top level items in the order i prefer)

If I later add more packages which add junk to the top level of my background menu, just calling (BkgMenu.fixup) again will hide anything new under " Exec " with the rest of the junk.

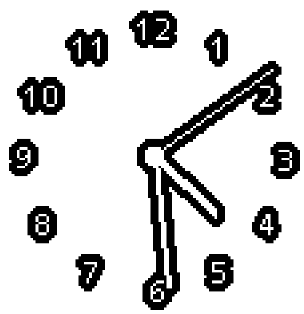
When any of the above functions (except BkgMenu.add.item) require you to specify an item, you can usually just give a string with the menu entry (or an atom, which is coerced to a string). The case has to be correct, and blanks have to be in the right place. The function will do a breadth first search of the background menu and all its submenus to find such an entry. If for some reason you have the same entry in more than one menu, you'll have to disambiguate it. To do this, you pass a list for the item, where the first thing in the list is the menu entry, and the rest of the list is a path through the tree to find it. For instance, the item (one two three) means find an entry whose text is "three", then find an entry in the tree underneath it whose text is "two", and the find an entry under that whose text is "one".

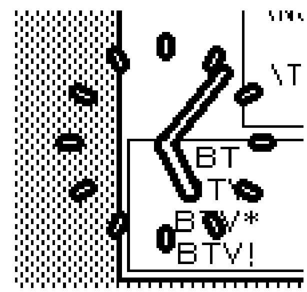
The item argument to BkgMenu.add.item is a standard menu item, i.e. a list of (label form help.string).

All of the functions return T if they were able to do as asked and NIL otherwise (you tried to do something with a menu entry which isn't there, or you tried to make a circular menu structure). The only exception to this rule is BkgMenu.subitems, which as previously mentioned returns a list of the subitems, or the atom NotAnItem if it's given a nonitem.

BICLOCK

By: Bernt Nilsson (Bernt Nilsson:Ida:LiTH or BKN%LiUIDA.UUCP@Seismo.ARPA)





INTRODUCTION

BICLOCK, a realtime screen clock, with hour, minute and second hands. It behaves as a kind of icon, in that it is partly transparent. The snapshot to the right above shows that the greater part of the window's area is transparent, in this case the gray background and part of the PSW window show through. The clock-image may be either black or white, and the shadow around the image-parts the opposite color: The reason for having a shadow is to make the clock easier to read. Markers and/or digits may be chosen.

Seconds

The seconds hand is optional. When on, the clock process consumes a lot of time, but because (1) it blocks very frequently and (2) shuts itself off temporarily, if the "load" is higher than usual, that does not seem to irritate at all. The "load" is measured by the continuous average of time spent in block, i.e. each round robin. A "usual load" is computed by a second level of average over the "load" value. The "usual load" is limited by upper and lower "reasonable" constants. If "load" is more than 10% above "usual load" then the seconds hand is shut off. This is a rather heuristic method, but works reasonably well. When the seconds hand is shut off, the image is updated approximately once per minute.

WINDOW COMANDS

Most window commands are applicable. The clock will preserve its "square" form when shaped. More details of the hour/minute markers/digits will appear only when the window is large enough, less if the window is smaller. The digits' fonts are chosen among fonts already "in core".

Left button

Left buttoning the window will print the current date (and alarmtime) in the promptwindow. Holding down left shift key while buttoning the window will copyselect current date, by BKSYSBUF.

Middle button, Alarm

There is a simple alarmclock facility built into the clock. That and some other options may be accessed by middlebuttoning the window and selecting the appropriate command from the menu that pops up.

Choosing the command "Set Alarm" will change the clock into showing current alarmtime and attach an adjust menu below the clockwindow. If no alarm time is known, current daytime plus 1 minute is used as default. Any change to the alarmtime is shown both by the clock and printed in the promptwindow. Buttoning "OK!" in the attached menu exits the adjust mode. When the alarmtime is reached, you will be aware of that fact... to shut the alarm off, use the "Alarm Off" in the popup menu.

FUNCTIONALITY

One clockwindow is usually created automatically when the file is loaded. More clockwindows may be created by the following function call:

(BICLOCK *props*) [Function]

Creates a new window. Value is the new window. The call may be done either with a free property list as first arg or spread. See properties below. The different properties are also controlled by the variables below. It is allowed to have more than one clock running, but see "bugs" below.

When the BICLOCK file is loaded, the following variables are initialized, some through INITVARS, so that you can set them before loading the file:

BICLOCKUSERPROPS [Variable]

Change this to customize the overall behavior of the clock. Value must be a free property list. Defaults to NIL.

BICLOCKDEFAULTPROPS [Variable]

These are the default properties, don't modify this, modify BICLOCKUSERPROPS above.

Properties are searched for in the following order.

1. those specified in the function call.
2. those specified by the BICLOCKUSERPROPS variable.
3. those specified by the BICLOCKDEFAULTPROPS variable.

BICLOCKINITIALPROPS [Variable]

Modify this if you like some specific property for the initial clock. BICLOCK is called with these properties when the file is loaded, defaults to NIL. If you want no initial clock, set it to (CREATE NIL).

BICLOCKWINDOW [Variable]

The window of the clock that is created when the file is loaded.

BICLOCKIDLEPROPS [Variable]

(IDLE.BICLOCK) [Function]

BICLOCK is called with BICLOCKIDLEPROPS from the Idle function IDLE.BICLOCK, defaults to start bouncing at center of screen.

Properties

Recognized properties and allowed values are:

SECONDS	T (default), if seconds hand shall be used, or NIL, if no seconds
COLOR	interior color, one of WHITE (default) or BLACK
MARKS	NIL (default), if no marks should be used, or one of HOURS, HOUR&MINUTE, 3/6/9/12 or a modulo number

DIGITS	NIL, if no digits, or one of HOUR (default), 3/6/9/12 or a modulo number
DIGITFORMAT	one of ARABIC or ROMAN
CHIME	NIL (default), no chime,or one of HOUR, QUARTER or a modulo number
ALARM	NIL (default), no alarm, or a the standard string representation of a date
SIZE	a number describing both width and height of the clock window, defaults to 119
HORIZONTAL	generic place on screen, one of LEFT, CENTER, RIGHT (default), or a number to specify left border of window
VERTICAL	generic place on screen, one of BOTTOM, CENTER, TOP (default), or a number to specify bottom border of window
REGION	if nonNIL, a region, overides SIZE, HORIZONTAL and VERTICAL for clockwindow
WINDOW	if nonNIL, a window, overides REGION, SIZE, HORIZONTAL and VERTICAL
CREATE	T (default), create a clock, or NIL, do not create, probably only useful if you want no initial clock...

The nonformal meaning of the Modulo numbers mentioned above are: "1" means: on each possible place. "2" means: on even places and so on...

KNOWN BUGS:

1. If the window is reshaped when in "set alarm" mode, the whole window group will be square.
2. Running more than one clock concurrently with seconds hand on, may behave a bit round robin because the load sensor is local for each clock and "senses" the other clocks.
3. Overlapping two clock windows creates a funny image in the intersecting area.

BITMAPFNS

By: Larry Masinter (Masinter.PA@Xerox.COM)

This document last edited on 4-mar-87

(READBINARYBITMAP *WIDTH HEIGHT FILE*) [Function]

reads a series of bytes from *FILE* and creates a *WIDTH* times *HEIGHT* bit map with contents. Note that each scanline of the bit map is rounded up to the nearest multiple of 16 bits (two bytes).

(WRITEBINARY BITMAP *BITMAP FILE*) [Function]

writes out *BITMAP* to *FILE* in format read by READBINARYBITMAP. Please note that READBINARYBITMAP must be supplied with width and height.

(WRITEBM *FILE BITMAP*) [Function]

writes *BITMAP* on *FILE* first preceding with width and height (in binary) such that it can be read in with READBM.

(READBM *FILE*) [Function]

reads width, height, and then appropriate size bit map.

(WRITEBMLST *FILE LST*) [Function]

writes a list of bit maps on *FILE.*

(READBMLST *FILE*) [Function]

reads a list of bit maps.

The following functions open and close *FILE.*

(READPRESS *PRESSFILE*) [Function]

reads press file *PRESSFILE* and returns a bit map. Can only handle press files generated by PRESSBITMAP and a couple of other utilities. Has no smarts, and is not easily extended.

(WINDOWBM *BITMAP POSITION*) [Function]

creates and returns a window containing image of *BITMAP*. Will be at POSITION or (GETPOSITION).

XEROX

Xerox Corportation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Bitmap-Gallery

Sampler and Documentation for bitmaps, especially those useful as screen background.

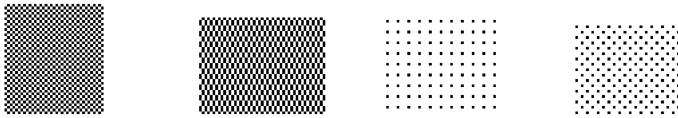
Eventually I'll rationalize the names and packages. Most of these bitmaps have been snapped from the screen and are smaller than they might appear. They usually look better on screen than they do on paper. Please send suggested additions to Foster.PA.

WARNING: This file is only usable in Lyric.

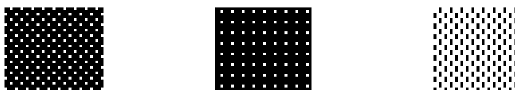
Author: Gregg Foster et al <Foster.pa>
Filed on: {FS8:PARC:XEROX}<FOSTER>LISP>USERS>BITMAP-GALLERY.TEDIT;5
Last filed: March 23, 1988 18:07 PDT (Wednesday)

Some Basic Shades

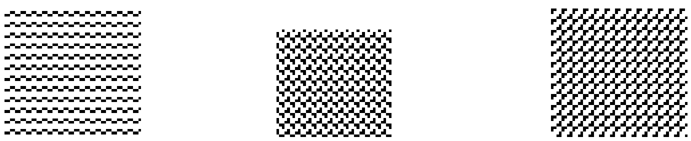
Use (IL:EDITSHADE) to create your own.



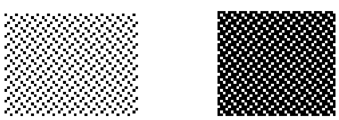
IL:PLAINSHADE, IL:GRAYSHADE, IL:GRAYSHADE1, IL:GRAYSHADE2



IL:GRAYSHADE3, IL:GRAYSHADE4, IL:DEFAULTSCREENSHADE

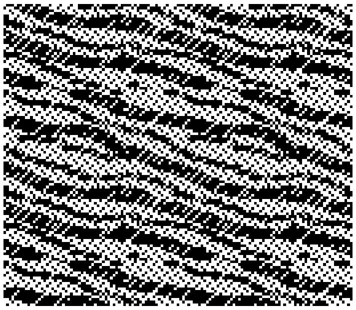


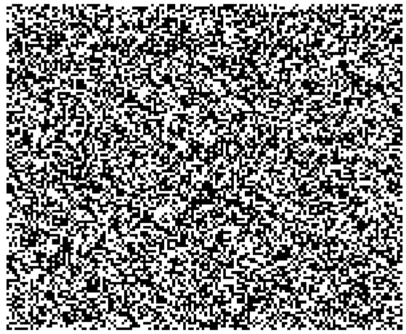
IL:WAVE-TEXTURE, IL:WAVE2-TEXTURE, IL:MESH-TEXTURE



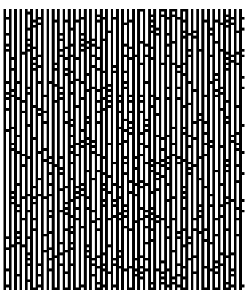
IL:DI-TEXTURE, IL:DARK-DI-TEXTURE

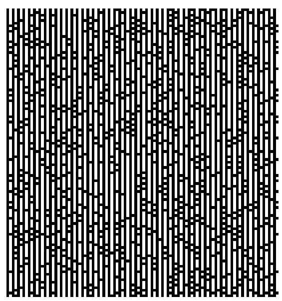
From Gregg Foster

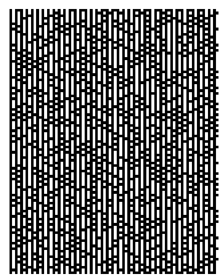




XCL-USER::*TESSEL-BM*, XCL-USER::*RANDOM-BM*





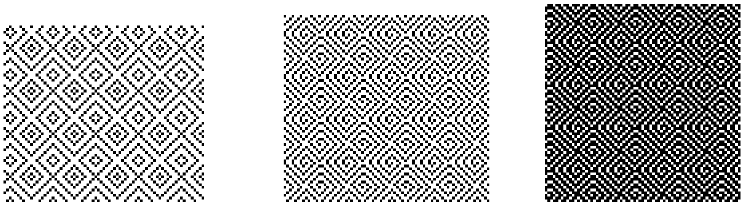


XCL-USER::*GRANITE-LIGHT-BM*, ...-MEDIUM-..., ...-DARK-...

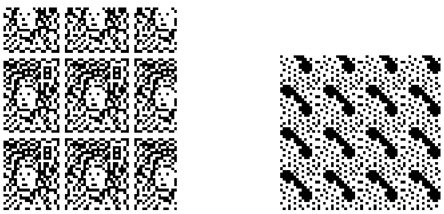
From {PHYLUM}<Foster>Lisp>Users>GRANITE, using the function il:|MakePseudoRandomBitmap| you get bitmaps that look something like the above (they come in three shades, LIGHT, MEDIUM, and DARK (symbols for 3, 2, 1, respectively); and large variety. The three bitmaps above are examples of the kind of thing you get.

[Example usage: (setq my-bitmap (il:|MakePseudoRandomBitmap| NIL 64 64 3), makes a 64x64 light PseudoRandom bitmap.]

From Stanley's Tool Works

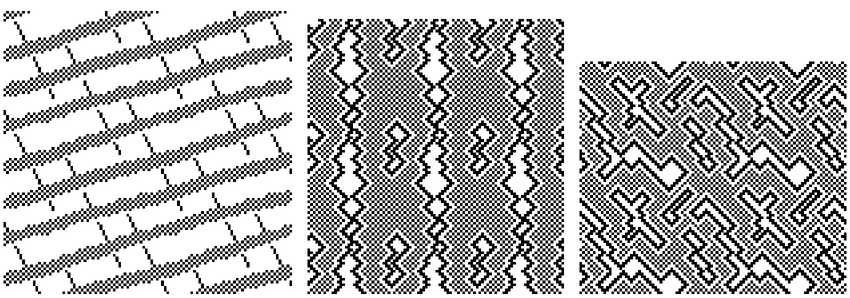


IL:LIGHTWALLPAPER, IL:WALLPAPER, IL:DARKWALLPAPER

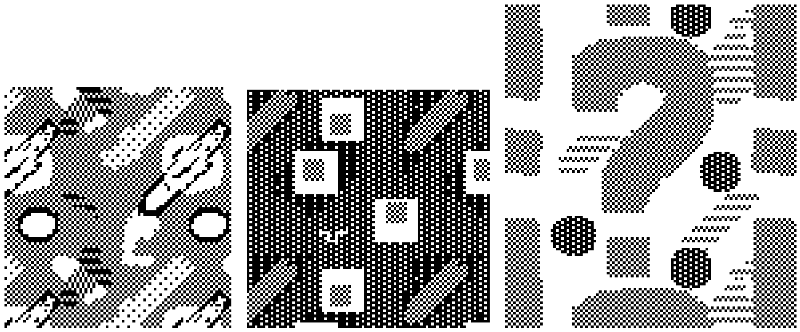


IL:*STAMP-BITMAP*,   IL:*PHONE-BITMAP*

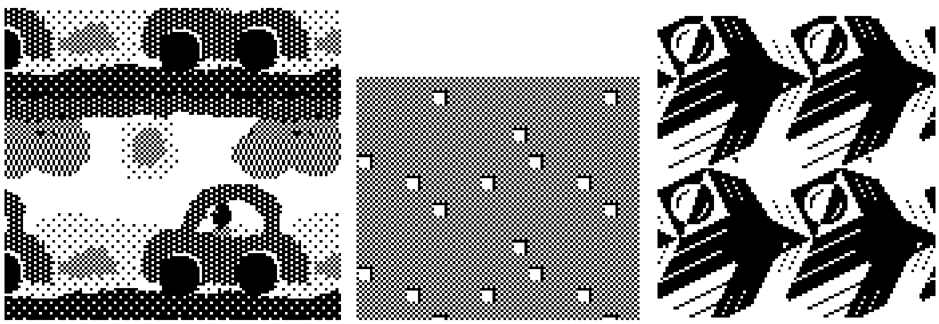
From Andreas Wickberg



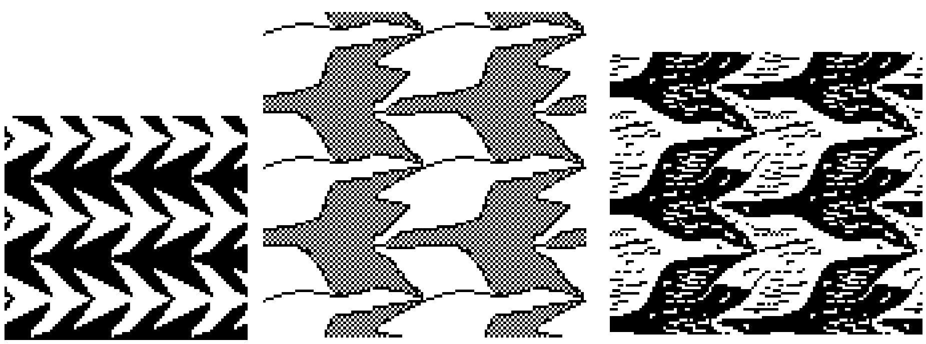
IL:AVANTBACKGROUND0, ...1, ...2



IL:AVANTBACKGROUND3, ...4, ...5

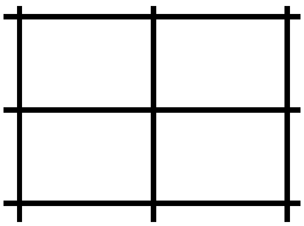
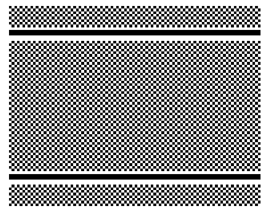


IL:AVANTBACKGROUND6, ...7, ...8

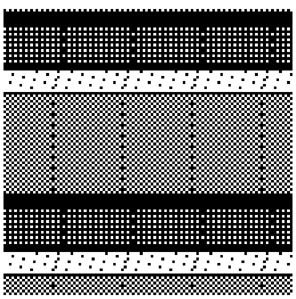
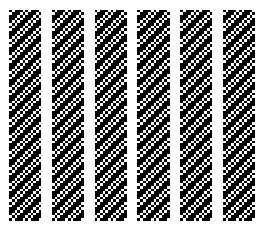


IL:AVANTBACKGROUND9, ...10, ...11

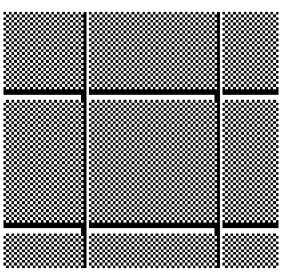
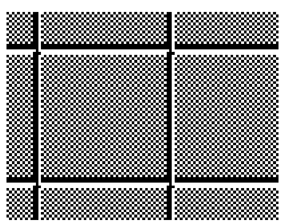
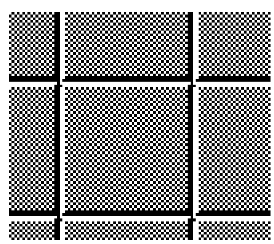
From Stu Card

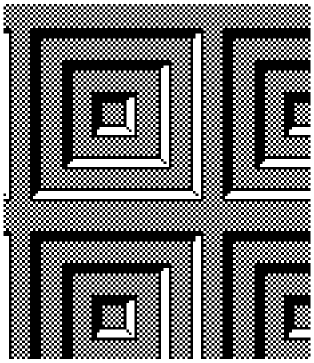
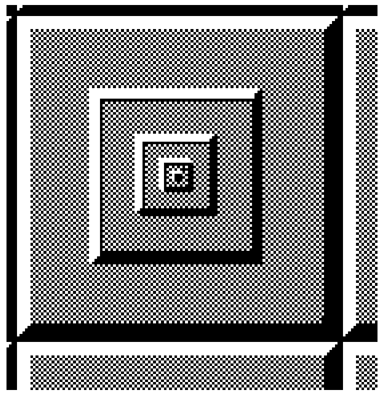
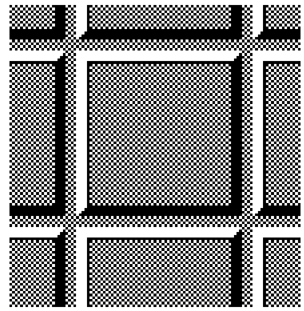
IL:ROOM.BM, IL:LINE1.BM, IL:LINE2.BM,

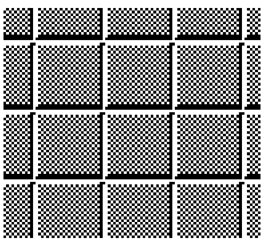
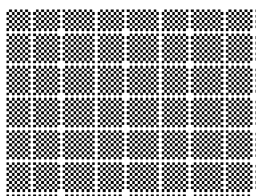
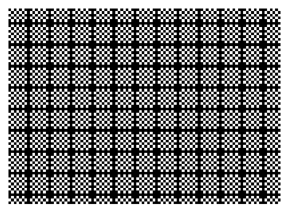
IL:LINE3.BM, IL:LINE4.BM, IL:LINE5.BM

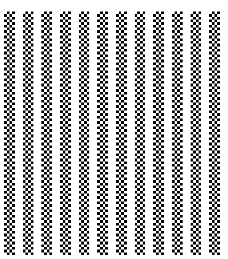
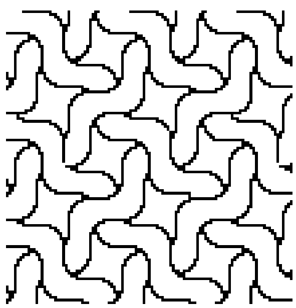
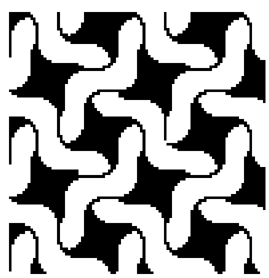
IL:SQUARE1, IL:SQUARE2, IL:SQUARE3.BM, ■

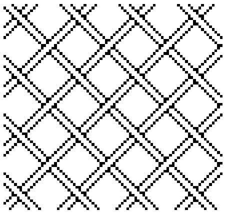
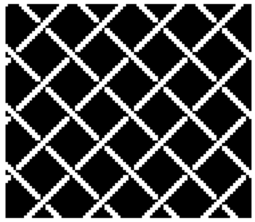
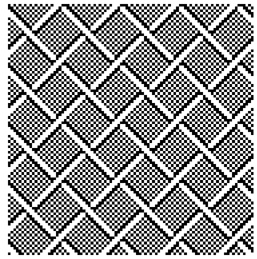
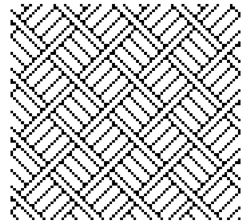
IL:SQUARE4.BM, IL:SQUARE5.BM, IL:SQUARE6.BM

IL:SQUARE7.BM, IL:SQUARE8.BM, IL:SQUARE9.BM

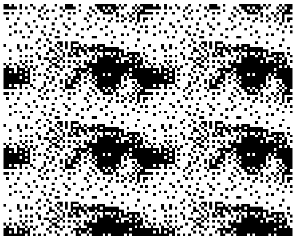




IL:SHIRT1.BM, IL:CURLY, IL:CURLY1

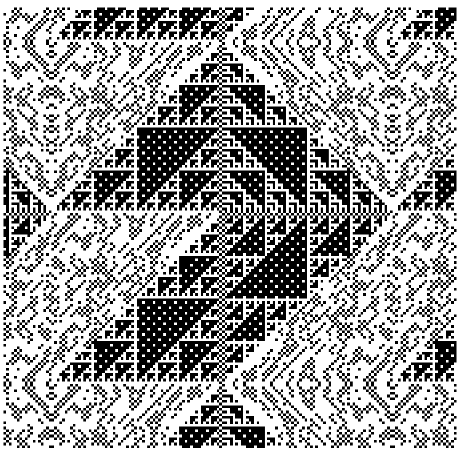
IL:WOVEN, IL:WOVEN1, IL:WOVEN2, IL:WOVEN3

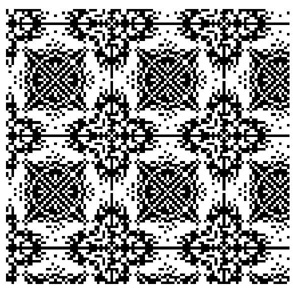
From Harley Davis



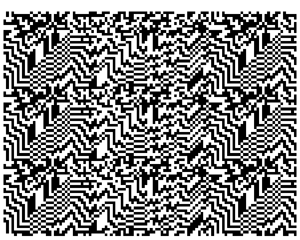
XCL-USER::*EYE-BM*

From John Corbett



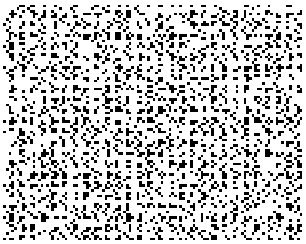
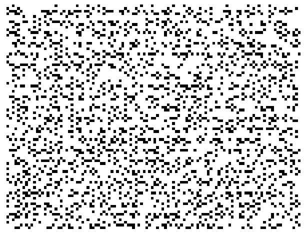


XCL-USER::*FRACT-BM*   XCL-USER::*MANDALA-BM*

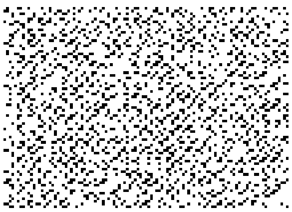
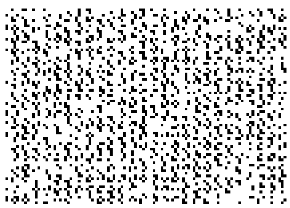


XCL-USER::*STATIC1-BM*   XCL-USER::*STATIC2-BM*

Fabrics (from John Corbett and Gregg Foster)

XCL-USER::*TWEED-BM* XCL-USER::*CHAMBRAY-BM*

XCL-USER::*CANVAS-BM*   XCL-USER::*CORDUROY-BM*

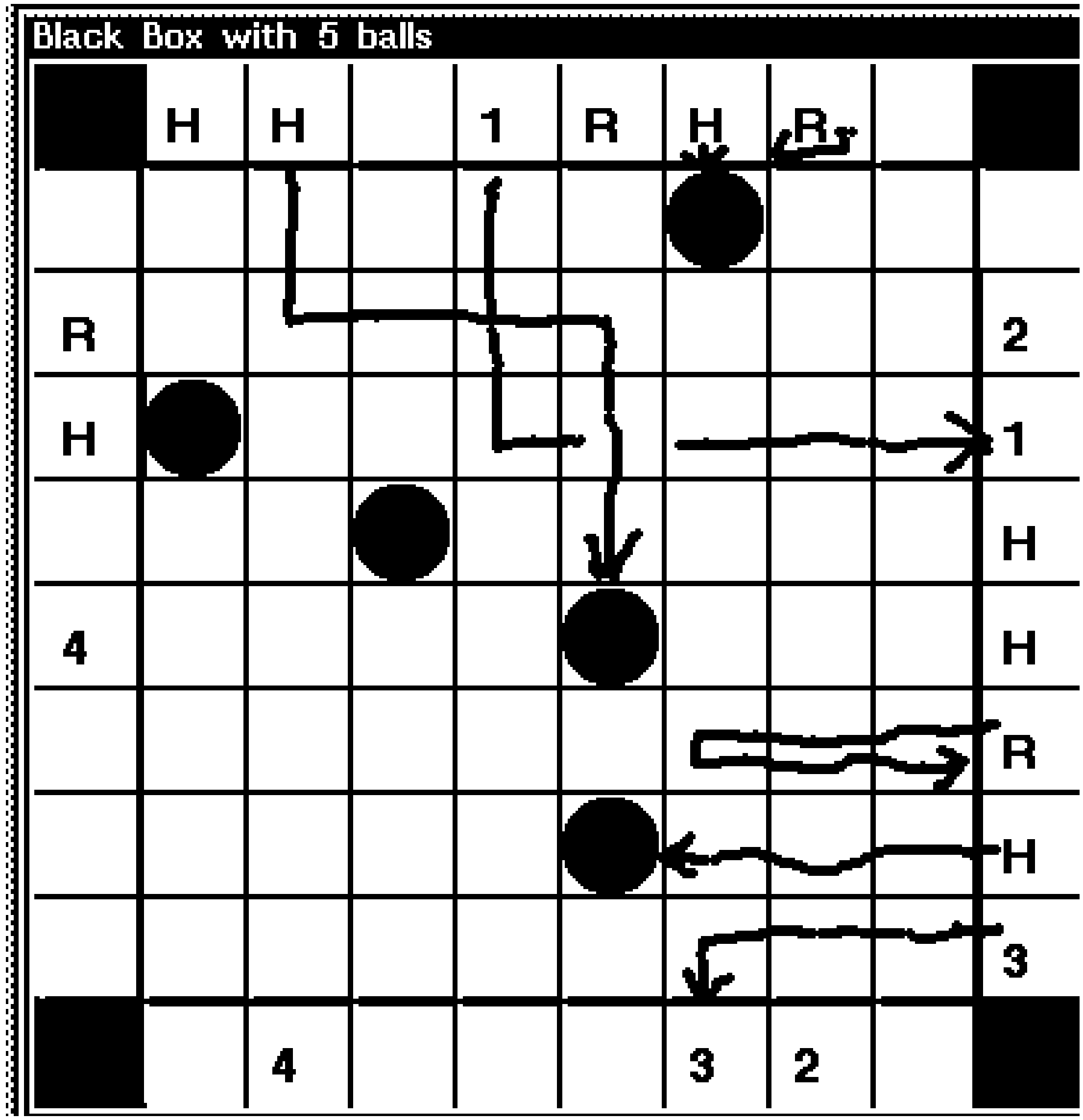
 

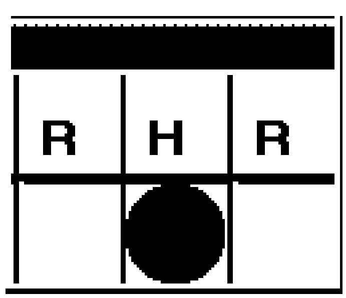
XCL-USER::*SEERSUCKER-BM*   XCL-USER::*BURLAP-BM*

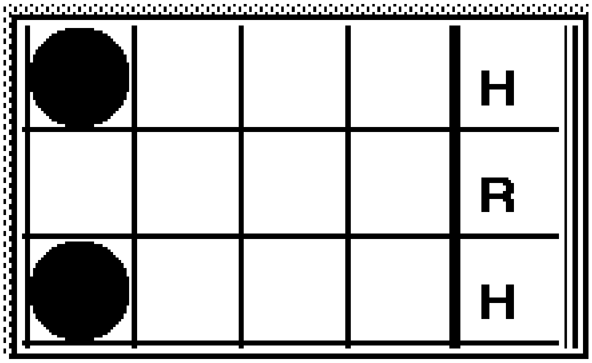
You can use the function FABRICIZE (included in this file) on an arbitrary bitmap to return a fabric-like bitmap (the original bitmap is unaltered).

USAGE: (FABRICIZE BITMAP)

Black Box
A Game



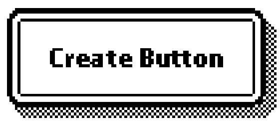



H
R
H

BUTTONS

Johannes A. G. M. Koomen
(Koomen.wbst@Xerox)

Created: November, 1986
Last Edit: December 2, 1988



DESCRIPTION

BUTTONS is a facility for creating icons which will trigger actions when they are clicked in. Each button has a label and an action associated with it. There are three different things which one can do with buttons: Trigger the action, move the button, and bring up a button command menu. These are initiated by use of the left, middle, and right mouse buttons within each button. The command menu is also available through the background menu entry "Button Control." The button world can be tailored (somewhat) using button properties.

Clicking with the Left mouse button on a button and then letting up causes the action associated with the button to be taken. If the action is a list it will be evaluated, otherwise it is stuffed into the system read buffer. The button inverts while the action is being taken.

Clicking with the Middle mouse button allows one to move the button on the screen. The button moves on a grid, unless the left shift key is down.

Clicking with the Right mouse button brings up a menu with the following commands:

Redisplay -- redisplay this button
Move -- same as clicking with the middle button
Copy -- make a copy of this button and move it
Edit -- invokethe structure editor on the label and the action of this button
Close -- close this button (but keep it)
Rollout: **Close All Buttons** -- close all open buttons
Delete -- delete this button
Rollout: **Delete All Buttons** -- delete all existing buttons
Create Button -- make a new button and move it

Expose Buttons -- redisplay all buttons (including previously closed ones)
Align Buttons -- prompts for alignment axis, then for successive buttons to line up
Save Buttons -- save current buttons and button properties in default data file
Rollout: **Save Some Buttons** -- prompt for file and for which buttons to save
Restore Buttons -- discard current buttons, restore saved buttons and properties
Rollout: **Load Some Buttons** -- prompt for file to load, keep/discard current buttons

When BUTTONS is loaded, a single "Create Button" button is placed in the lower left corner of the screen. See RESTORE.BUTTONS below for setting up your buttons programmatically.

FUNCTIONAL INTERFACE

(CREATE-BUTTON *action label location noopenflg*) [Function]

Creates a button with indicated action and label at the given location and displays it unless *noopenflg* is non-NIL. If *action* is NIL, *label* will be used for *action*. If *label* is NIL, (CAR *action*) will be used for *label* if *action* is a list, *action* otherwise. If both *action* and *label* are NIL, the values of the button properties DEFAULT-ACTION and DEFAULT-LABEL will be used instead. If *location* is not a POSITION or a REGION, the user is prompted for a location.

(BUTTONSPROP *propname {newvalue}*) [Function]

Returns the current value of the button property *propname*. If *newvalue* is given, it becomes the new value. The following properties (and their initial values) are currently in use :

DEFAULT-LABEL	"Create Button"
DEFAULT-ACTION	(CREATE-BUTTON)
MENU-FONT	(MODERN 12 BOLD)
LABEL-FONT	(MODERN 10 BOLD)
GRID-ORIGIN	(15 . 15)
SAVE-DIRECTORY	NIL
EDIT-SHADE	4104
EXEC-SHADE	65535

The value of SAVE-DIRECTORY must be acceptable to the function DIRECTORYNAME (i.e., either NIL for login host & directory, or T for current directory, or a standard host & directory spec).

(RESTORE-BUTTONS *filename keep-current-buttons?*) [Function]

Reinstalls the buttons stored on *filename*, which defaults to SAVED-BUTTONS.DATA on the directory indicated by the button property SAVE-DIRECTORY. Existing buttons are discarded unless *keep-current-buttons?* is non-NIL.

(SAVE-BUTTONS *filename buttons*) [Function]

Saves the given *buttons* in *filename*, which defaults to SAVED-BUTTONS.DATA on the directory indicated by the button property SAVE-DIRECTORY. If *buttons* is NIL, all current buttons will be saved.

(SAVE-SOME-BUTTONS *filename buttons*) [Function]

Saves the given *buttons* in *filename*. If *buttons* is NIL, you are prompted to indicated the buttons to be saved. If *filename* is NIL, you are prompted to supply a file in which to save the indicated buttons.

(LOAD-SOME-BUTTONS *filename*) [Function]

Loads the buttons in *filename*. You are prompted to indicate whether to keep the current buttons or discard them. If *filename* is NIL, you are prompted to supply a file from which to load the buttons.

CALENDAR

By: Michel Denber (Denber.WBST @ Xerox.COM)

Uses: TABLEBROWSER, TEDIT

INTRODUCTION

CALENDAR is a program which can be used to display a calendar on your screen, and keep track and remind you of events and appointments. Calendar 2.04 (the current distributed version) runs in the Koto or Lyric releases of Lisp. The version number appears in the title bar of each Calendar window. Calendar needs the Lisp Library package TABLEBROWSER, which it loads automatically. It also uses TEdit. Various font sizes (from 8 to 36) in the families TimesRoman and Helvetica may be needed, depending on the size chosen for month windows. Reminder files created by earlier versions of Calendar are incompatible with this version.

I. STARTING CALENDAR

Load CALENDAR.LCOM from your favorite LispUsers directory, eg.

LOAD ({ERIS}<LISPUSERS>CALENDAR.LCOM]

and then type (CALENDAR). You will get a menu of years (the menu always shows five years starting with last year). If you select a year with Left, it will create a Year window containing a calendar for that year. Each month in the Year window is also a menu item. If you now select a particular month with Left, CALENDAR will create a Month window showing a calendar for that month. You can now select a particular day within the month to bring up a Day browser (described in the next section). The Month window also shows small calendars of last month and next month. You can bring up those months in the current month window by selecting them with Left. If you select them with Middle, the program will create a new window for that month. The Year menu has an entry labelled ‘‘Other". If you select this, it will prompt you to type in a year, if you want one that isn’t on the menu.

You can have as many year and month windows open at the same time as you like. Month and day windows can also be reshaped to occupy less room on the screen. You can Shrink any of the CALENDAR windows to an appropriate icon, or close them when they are not needed. The reminder facility remains active. If you close your last year window, call (CALENDAR) again to get a new one. CALENDAR uses the Lisp Prompt Window to display informative messages.

Please send your comments, suggestions, and bug reports to me - Denber.WBST (ARPA: Denber.WBST@Xerox.COM). Thanks.

II. REMINDERS

The Day Browser

Clicking Left in any day in a month window will open a browser on that day. The browser displays each reminder for the day, along with its event time if it is a timed reminder. You may have more than one browser open at the same time. When you close a month window, it will automatically close all day browsers for that month. There is a menu across the top of the browser with the following items:

Add: Lets you create a new reminder in this day. If you select Add, the program will bring up a TEdit window containing a template for the new reminder. The template contains several fields you can select and fill in. These are described in Creating Reminders, below.

Display: Brings up the full contents of the reminder in a TEdit window.

Delete: Useful for deleting reminders that you no longer need. By default, timed reminders are deleted automatically after they "fire"; untimed reminders do not fire and are never deleted automatically. Calendar will immediately remove reminders which you delete from the month window (and the reminder's line in the day browser is crossed out), however it will leave reminders that have fired visible in the month window until you redisplay it (eg. September is visible, you select Redisplay from the right-button menu in the title bar or select September again in the Year window, and all fired September reminders will be purged from the month window when it redraws).

Update: Saves your reminders to disk (see the section on Saving reminders below).

Send Mail: Prompts you for a name to send to. The selected reminder will be mailed to that person when it activates, rather than displaying on your screen. Note that no validity checking is done when you enter a name, so your message could conceivably not be delivered if you typed the name wrong, for example. The message is mailed when the time arrives. Of course, this assumes that your system is running at that time, that you have Lafite active, and that Lafite is running in the mode (GV or NS) corresponding to your intended recipients.

Period: Brings up a menu with the choices Daily, Weekly, Monthly. The selected reminder will be made periodic and will appear at the selected intervals.

Creating Reminders

You can create a new reminder either by clicking Add in a day browser, or by clicking the middle button in a day box in the month window. This opens a new reminder form with the following fields:

Title: The reminder title should not exceed one line in length. This field will be displayed in the Day browser and the month window. This field may not be omitted; all others are optional.

Event time: The scheduled time for the event. By default, this is also the time at which the reminder will be activated. If this field is omitted, the reminder is "untimed". Untimed reminders do not alert you. When a timed reminder activates, it beeps and brings up a TEdit window containing the full reminder text.

Alert time: The time at which you would like the reminder to activate. You might want to be reminded of a meeting 10 minutes early, for example. The alert time can be set to any time, before or after the event time, as long as it is in the same day. If this field is omitted, it defaults to the value of the event time.

Alert: Edit this field to contain just the word Yes or No. If you choose No, the reminder will not alert you, even if it is a timed reminder. If this field is omitted, it defaults to the value set in the Options menu (see Programming below).

Duration: The expected length of the event. Version 2.04 makes no use of this field.

Message: The actual message you want to save. This may be any TEdit text or omitted entirely.

The new reminder form includes a menu with the choices Save and Abort. After filling in the fields you want, clicking Save will add the reminder to the system and close the form. Clicking Abort at any point cancels the reminder being created.

The time can be entered in almost any reasonable format, eg. 9:00 AM, 9 AM, 9 a.m., 2:30 PM, 2:30 P.M., 1430, or can be left out by skipping over the field. Times are "AM" by default, so if you only type 8:30, it will assume 8:30 AM. A heuristic is included to ask "Are you sure?" if you type a time earlier than 9 without an AM/PM qualifier (this value is controlled by CALDAYSTART, see Programming, below). Times of noon and midnight are special cases. There is no generally accepted meaning for the expressions "12:00 AM" and "12:00 PM". If you want a reminder at noon, enter the time as "12:00" or just "1200". Because reminders are added to a particular day, midnight is ambiguous; there is no provision for entering a time of midnight.

If you add a reminder for a time that is already in the past (for example, to keep a historical record of an event after the fact), the program will save the reminder but will warn you that the reminder time has already passed.

Expired timed reminders are automatically deleted upon expiration by default. Setting the variable CALKEEPEXPIREDREMS (see Programming, below) will cause timed reminders to be retained after firing.

Reminders which are scheduled for a time when your machine is not running will not be activated the next time you login. This avoids having a possibly long sequence of "dead" reminders popping up at login time.

Saving and loading reminders

You can save your reminders in a file at any point. The first time tou start Calendar , it will ask you to provide a default host and directory for reminder files. You should enter this in the usual format, for example {DSK}<Lispfiles> or {ERIS}<your-name>LISP>. This will become the new value of CALDEFAULTHOST&DIR (it is initially NIL). To save your reminders, select Update from any day browser. This will open a pop-up menu of currently loaded files, plus an "other" item for giving a new file name. If you enter a new name, all currently unsaved reminders will be stored under that name. If you select an existing file, the contents of that file will be updated and any new reminders created since the last update will be added to it. If you abandon your sysout or if your machine crashes, you can have Calendar automatically reload your reminders file when you restart (see CALDEFAULTHOST&DIR and CALLOADFILE in Programming, below). You can also load a reminder file at any time by holding the middle button down in the title bar of a month window. This will open a pop-up menu of files that have already been loaded, plus an "other" item to specify a new file. In this version of Calendar there is never any need to load a reminder file more than once. The menu is useful, however, to show which files have already been loaded.

An "almanac" reminder file is distributed along with Calendar. It contains a variety of holidays and notable dates for the year. The file is called CALMANACnn, where nn is the last two digits of the year. For example, the file for 1986 is called CALMANAC86. You can load this file by selecting Other from the middle button menu and typing CALMANAC86.

By default, the program will only save your reminders when you select Update. You may control file updating by changing the Auto File Update option available under the Options menu item in the month window. See Programming, below.

III. PROGRAMMING

A programmatic interface is provided to let you create day, month, or year windows from your own programs.

If your reminder text is a Lisp list (anything inside parentheses), when the reminder fires the program will evaluate the list rather than displaying the reminder in a window and beeping.

Functions

(CALENDAR *m d yr*) [Function]

m, d, and yr are integers specifying a month, day, and year, respectively. Arguments are specified as follows:

If only yr (must be 4 digits) is supplied, brings up a year window for that year and returns yr.

If m and yr are supplied, brings up a month window for that month and returns m.

If m, d, and yr are supplied, brings up a day window for that day and returns d.

For invalid combinations (missing yr, d and yr only), returns NIL. Also returns NIL if yr is out of range (the calendar algorithm is only valid for years between 1700 and 2100).

Examples:

(CALENDAR NIL NIL 1984) shows a calendar for 1984 and returns 1984.

(CALENDAR 10 NIL 1984) shows a calendar for October 1984 and returns 10.

(CALENDAR 10 NIL 84) returns NIL (out of range).

(CALENDAR 10 21 1984) shows October 21st, 1984 and returns 21.

You can also call Calendar with the keywords TODAY, THISMONTH, and THISYEAR.

Examples:

(CALENDAR ’THISYEAR) shows a Year window for 1986, if this year is 1986. This might be used in an init file, to always start a Calendar of "this year".

(CALENDAR ’TODAY) opens a Day browser for today, containing all of today’s active reminders.

(CALLOADFILE *file-name*) [Function]

Loads the file *file-name* into the reminder system and returns T. Returns NIL if the file is not found or is not a valid reminder file.

Example:

(CALLOADFILE ’{DSK}<LISPFILES>CALREMINDERS)

Variables

CALALERTFLG [Variable]

Initially T. This controls whether or not reminders whose Alert field is not specified should alert you when they fire. T means they will. NIL means they won’t.

CALDAYDEFAULTREGION [Variable]

Initially (32 200 350 100). This specifies the default size for day browsers. The location is only used for day browsers opened programatically.

CALDAYSTART [Variable]

Initially 900. This represents the time (in 24 hour format) at which your regular day starts. The system will use it to confirm times you enter without a "PM" indicator if they are less than this value. For example, it is more likely that 4 means 4 PM than 4 AM.

CALDEFAULTALERTDELTA [Variable]

Initially 0. This represents the time (in minutes) before or efater the event time you want reminders to be activated, if no explicit alert time was given for them. To be reminded before the event, make this value negative. The resulting time must still be in the same day as the event.

CALDEFAULTHOST&DIR [Variable]

Initially NIL. This is the host and directory on which your reminder files will be saved if you type the file name without a directory specification. The system will prompt you to enter a value for this the first time you start it.

CALFLASHTIMES [Variable]

Initially 0. Specifies the number of times to flash the destination given by CALFLASHTYPE when a reminder is activated.

CALFLASHTYPE [Variable]

Initially ’None. Specifies which window should be flashed when a reminder is activated. Can be set to ’WINDOW, to flash the reminder display window, or ’SCREEN to flash the entire screen. CALFLASHTIMES (above) should be set to the desired number of flashes.

CALFONT [Variable]

Initially ’TimesRoman36. This variable controls the font used to display the Month Window. You can change it for example, by saying (SETQ CALFONT (FONTCREATE ’HELVETICA 18)). The change takes effect the next time you display a month. If you reshape a month window, the program will try to find a smaller font to fit the new window size, but the value of CALFONT will not be changed.

CALHARDCOPYPOMFLG [Variable]

Initially T. This variable controls the printing of the phase-of-the-moon icons when you hardcopy a month window. Setting it to NIL suppresses this printing. Month windows are hardcopied at printer resolution in Koto, screen resolution in Lyric.

CALHILITETODAY [Variable]

Initially ’CIRCLE. This variable determines how today’s date will be highlighted in a month window. The default is to draw a circle cround it. If you set this to ’BOX, a light gray grid will be placed over the date. Setting this to NIL suppresses all date highlighting.

CALKEEPEXPIREDREMSFLG [Variable]

Initially NIL. If you set this to T, Calendar will not automatically delete reminders when they fire (they can still be deleted using the Delete menu command, above). The default action is to delete reminders when they fire, although they will remain visible until the window is redisplayed.

CALMONTHDEFAULTREGION [Variable]

Initially (32 32 868 700). This specifies the default position and size for month windows. If you set the size to a value small enough to allow several month windows side by side, the windows will tile left to right, bottom to top.

CALREMDISPLAYREGION [Variable]

Initially (200 400 300 400). This specifies the default position and size for reminder display windows.

CALTUNE [Variable]

When a reminder is activated, it will play the tune stored here (in PLAYTUNE format).This is initially a two-note "ding-dong". Set this to NIL if you want no audible warning. 1100's and 1132's have no hardware for sound.

CALUPDATEONSHRINKFLG [Variable]

Initially 'Never. This means that Calendar will save your reminders on a file only when you explicitly click Update from a Day Browser. If set to 'Shrink, it will cause Calendar to save your reminder file automatically only when you shrink the Month window. This is useful when you are entering many reminders at the same time, but it means you must remember to explicitly shrink the month window or your reminders will be lost if your machine dies. If set to 'Always, causes Calendar to immediately save each reminder as soon as it is created.

You can also set these variables interactively by clicking on the box marked "Options" in any Month window. This brings up a freemenu similar to the TEdit expanded menu.

Alert: Specifies the default for the Alert field in the new reminder form. Sets the value of CALALERTFLG (described above).

Keep expired rems.: If set to No, the system will automatically delete reminders when they fire (although they remain listed in the month window until the next time you redisplay it). Sets the value of CALKEEPEXPIREDREMSFLG.

Auto. file update: Always means that the system will update the reminder file every time you create a new reminder. Shrink means update only when a month window is shrunken. Never means updates will be done only when you explicitly select Update from a Day browser. Sets the value of CALDUPDATEONSHRINKFLG.

Alert delta: Sets the value of CALDEFAULTALERTDELTA.

Host & dir.: Sets the value of CALDEFAULTHOST&DIR .

After you have made the selections you want, click Apply! This sets the selections and closes the menu. If you don't want to make any changes, just close the menu (like closing any window). This preserves the previous settings even if you changed them in the menu. Any changes you make to these variables are not saved automatically in reminder files.

IV. LIMITATIONS

Day groups must begin and end in the same month.

The calendar algorithm is valid only for years between 1700 and 2100.

V. KNOWN BUGS

Today-circling function occasionally fails to erase the old day.

VI. FUTURE PLANS

Automatic scheduling.

Automatic communication with other Calendars.

CANVASCONVERTER

By: Stephen Knowles (Stephen Knowles:49/89/636/13:Siemens AG)

Partly based on work by:
Matthias Schneider-Hufschmidt (Matthias Schneider-Hufschmidt:ZTISOF:SIEMENS)
Giselbert Schramm (Giselbert Schramm:ZTISOF:SIEMENS)

Uses: BITMAPFNS

This document last edited on 19-Sep-1988 13:32:21.

INTRODUCTION

This module enables the transfer of bitmaps between the Envos Lisp and Xerox ViewPoint environments. The medium used for the transfer is an NS file server (i.e. a file drawer which can be accessed by both environments). The possibility of transferring Lisp bitmaps into the ViewPoint environment is particularly useful for documenting Lisp applications.

MODULE EXPLANATIONS

There are essentially two major functions:

(IL:WRITECANVAS *BITMAP FILE*) [Function]

This function writes the *BITMAP* on to *FILE* and makes *FILE* of type ViewPoint Canvas, whereby *FILE* must be on an NS file server.

(IL:FETCHCANVAS *FILE*) [Function]

This function reads *FILE* into a Lisp bitmap, whereby *FILE* must be on an NS file server.

Additionally there are two auxiliary functions to aid in the use of the above two functions.

(IL:SNAPBM) [Function]

and

(IL:CANVAS-FROM-WINDOW *WINDOW FILE*) [Function]

EXAMPLES

All examples must be typed into an INTERLISP exec.

To write a canvas of a Lisp screen region:

```
(WRITECANVAS (SNAPBM)
```

```
 '{NSFileServer:Domain:Organization}<FileDrawer>Folder>TESTFILE)
```

To write a canvas of a Lisp window:

```
(CANVAS-FROM-WINDOW (WHICHW)
 '{NSFileServer:Domain:Organization}<FileDrawer>Folder>TESTFILE)
```

To read a canvas into a Lisp bitmap:

```
(SETQ  X (OPENSTREAM
'{NSFileServer:Domain:Organization}<FileDrawer>Folder>TESTCANVAS 'INPUT))
(EDITBM (SETQ LISPBITMAP (FETCHCANVAS X)))
(CLOSEF X)
```

CAVEAT

When fetching a canvas, there is a 50-50 chance that the Lisp bitmap will be O.K. It could, however, come out distorted (this is due to the differing ways in which ViewPoint and Lisp handle bitmaps, Lisp uses 16 complement, ViewPoint 32 complement - or something like that). If this should be the case, simply increase the canvas width in ViewPoint by 5 millimeters (approx. 16 pixels) and repeat the fetching process.

Unfortunately in the Lyric version if one repeatedly wrote a canvas with the same name, the file server somehow got mixed up and set the file-info of the folder above the canvas into ”type = canvas”! One could put this right with the (SETFILEINFO...) function in Lisp, although under normal circumstances one does not write out a canvas repeatedly with the same name any way. I have been unable to test the behaviour in MEDLEY.

Compatibility has only been tested up to ViewPoint 1.1.

CD

By: Henry Thompson (HThompson.pa@Xerox.com)

This document last edited July 6, 1988.

INTRODUCTION

The file CD implements a UNIX*-style facility for manipulating the connected directory. It also insures that the connected directory is always displayed.

CD *PATTERN* [Exec command]

MODULE EXPLANATIONS

CD is defined as a command which allows low-overhead means of effecting many common changes of connected directory. Its behaviour is partly conditioned by three global variables:

CD.DEFAULT.HOST [Variable]

CD.DEFAULT.PREFIX [Variable]

CD.DEFAULT.USER [Variable]

CD.DEFAULT.HOST defaults to DSK. CD.DEFAULT.PREFIX defaults to the name (e.g. DSK) of the local disk volume on a Dandelion, otherwise NIL. CD.DEFAULT.USER defaults to the value of USERNAME, and is updated automatically after GREETing.

The value of CD is always a CONS-pair of the old and new connected directories.

On hosts which support some form of sub-directory, CD needs to know the character which is used to separate sub-directories. The table CD.OS.SEPRS is an a-list which determines this mapping - it is initialised to map UNIX* and VMS to "/" and DSK, NS and IFS to ">". To enter this table it looks up the host first in CD.OS.SEPRS directly, then via NETWORKOSTYPES. In the documentation which follows, ">" means whatever the separator is for the relevant host.

The possibilities for *pattern* are as follows:

empty

Connects to the directory determined by the conjunction of CD.DEFAULT.HOST, CD.DEFAULT.PREFIX and CD.DEFAULT.USER.

{anything

Interprets *pattern* as a complete directory specification, and connects to it.

<anything

Interprets *pattern* as a directory specification to be qualified by CD.DEFAULT.HOST and CD.DEFAULT.PREFIX, and connects to it. For example if CD.DEFAULT.HOST is {server} and CD.DEFAULT.PREFIX is NIL, then CD <dir>sdir> is equivalent to CD {server}<dir>sdir>, whereas if

CD.DEFAULT.PREFIX was /user and server was known to be running UNIX[*], then CD <dir/sdir> would be equivalent to CD {server}</user/dir/sdir>.

.>rest

Equivalent to CD rest. This is purely for compatability with UNIX[*].

..>rest

Equivalent to peeling off one (sub-)directory from the currently connected directory, followed by CD rest. For example, if connected to {server}<dir>sdir>, then CD ..>sdir1 is equivalent to CD {server}<dir>sdir1>. Note that because of common lisp reader pecularities, you cannot use .. alone under a common lisp read-table. The synonym << can be used instead.

otherwise

Treat *pattern* as a further specialisation of the current directory, and connect to the resulting sub-directory. For example, if connected to {server}<dir>sdir>, then CD ssdir is equivalent to CD{server}<dir>sdir>ssdir>.

Note that throughout, the closing ">" is optional.

Menu Interface

At any time you can left button in the window displaying the current connected directory, and see a menu of all the directories you have yet been connected to. Selecting one will move you there. You can also shift-select out of this menu into the current input stream. This latter is very useful when typing file names.

Middle buttonning in the directory display window will give you a menu of directories, followed by a menu of Connect/Browse/Delete. Connect does so, Browse brings up a file browser and Delete removes the directory from subsequent menus.

---

[*]UNIX is a trademark of Bell Laboratories.

CHATEMACS

By: Randy Gobbel (Gobbel.pa)

requires:CHAT, chatemacs.elc (GnuEmacs Lisp program)

This document last edited on August 24, 1987

INTRODUCTION

ChatEmacs, in conjunction with the **chatemacs.elc** module for GnuEmacs, enables use of the mouse for scrolling and selection in GnuEmacs. It also allows use of the META key for escape-prefix commands and automatically switches Chat in and out of Emacs mode when entering and leaving the editor.

DETAILS

After loading ChatEmacs, typing META-char will send an ESCAPE character, followed by the vanilla character. CTRL-META-char sends an ESCAPE followed by CTRL-char. Once ChatEmacs is active, most Emacs commands should require only one keystroke. Since Emacs was originally designed for terminals with a META shift key, this makes the Emacs command set somewhat more regular and easier to remember. For example, scrolling forward and backward will be on CTRL-V and META-V, respectively.

In order to enable mouse actions, first load CHATEMACS.LCOM into Interlisp. After opening a Chat connection and running GnuEmacs, either load chatemacs.elc manually (by giving the ^Xload command), or add the following line to your GnuEmacs init file (.emacs):

```
(load "chatemacs")
```

After loading chatemacs.elc, the title bar on your Chat window should say "Emacs ON". If not, middle-buttoning the "Emacs" menu item in your Chat window will enable mouse events to be sent to Emacs. After ChatEmacs has been activated for the first time, the Chat window’s title bar will always indicate whether Emacs mode is on or off. If your mouse clicks don’t seem to be taking effect, check the title bar first!

Automatic switching frees the user from having to manually turn ChatEmacs on and off when using Emacs. In most circumstances (see exceptions below) automatic switching will not interfere with other Chat operations, and can be left enabled. Auto-switching is controlled by:

CHATEMACS.SWITCH.ENABLED [Variable]

When this variable is non-NIL, Chat will respond to a sequence of two consecutive ESCAPEs by toggling the flag that controls mouse event sending. The state of the flag is noted in the window’s title bar, just as if the menu command had been executed. CHATEMACS.SWITCH.ENABLED is defaulted to NIL.

auto-switch-enabled [GnuEmacs Variable]

This variable controls auto-switching on the GnuEmacs side of the Chat connection. If it is non-nil, GnuEmacs will send a switch command when chatemacs.elc is loaded, and another when exited via a ^X-^C command.

Using Emacs with the mouse

The chatemacs.elc module, at the GnuEmacs end of the connection, determines the interpretation of mouse clicks. The current user interface is more complicated than I would like, and suggestions for improvements are welcome.

The most basic operations are fairly simple: left button in a text buffer moves the Emacs "point" to wherever the cursor is pointing. Right button moves the mark (the typein cursor will move for a couple of seconds just to show you where you've just put the mark), and copies the new region to the kill buffer (for use with "shift-select," see below).

Scrolling with the mouse works more or less as in Interlisp, with the scrollbar being the right-hand part of the screen past column 80. Alternatively, holding down the META key makes the entire text area act as a "scrollbar". As in most Envos environments, left button scrolls the line that the mouse is pointing to to the top of the window, right button moves the top line down to the mouse cursor, and middle button "thumbs", taking the vertical displacement of the mouse cursor as an offset into the file (i.e., top line = beginning of file, bottom line = end of file).

Shift- and control- mouse clicks perform editing operations: shift-left copies the contents of the kill buffer to wherever the mouse is pointing (the closest thing to Interlisp shift-select I could come up with). Control-left and control-right kill from point to where the mouse is pointing (sort of like control-select). Control-shift-left moves the mark without copying anything to the kill buffer.

Mouse clicks in the mode line and minibuffer do things that were inherited from il-mouse's ancestor, a package for the BBN Bitgraph terminal. Maybe you will find them useful. They are: The modeline acts like a sideways scrollbar, left=top. In the minibuffer, left button is equivalent to typing META-X, middle button evals an expression you type in, and (beware!) right button suspends Emacs (equivalent to typing ^X^Z).

As mentioned above, the current user interface is sort of, how shall I say, "gnarly." If you have better ideas, please let me (Gobbel.pa) know.

CHATSERVER

By: Larry Masinter (Masinter.PA@Xerox.COM)

This document last edited on September 7, 1988.

REQUIREMENTS

CHATSERVER-NS Requires: CHATSERVER and COURIERSERVE.

CHATSERVER-RS232 requires: CHATSERVER and (DLTTY or DLRS232C). As of this date, CHATSERVER-RS232 hadn't been tested with Medley.

CHATSERVER-TCP requires: CHATSERVER and TCP. As of this date, CHATSERVER-TCP is unreliable: the chat server sometimes leaves open the connection and will not open another one.

In general, a protocol chatserver requires CHATSERVER and a a protocol converter. Sources for TCP server available.

CHATSERVER also loads LispUsers modules CL-TTYEDIT and SIMPLECHAT.

The module PREEMPTIVE is useful in conjunction with CHATSERVER but not required.

INTRODUCTION

CHATSERVER is a general facility that allows a Lisp workstation to be controlled from a dumb terminal. In addition to CHATSERVER, you will need a protocol driver: something that connects the CHATSERVER to a communication protocol. The various protocol drivers are the mechanism by which CHATSERVER can be controlled; versions include using XNS via CHATSERVER-NS, TCP/IP TELNET protocol via CHATSERVER-TCP, and RS232 via CHATSERVER-RS232. CHATSERVER-NS is the most reliable, although CHATSERVER-RS232 has worked reliably in the Lyric release.

The server implements password protection using the same mechanism as IDLE. There is another variable, CHATSERVER.PROFILE, which gets searched first for ALLOWED.LOGINS so that you can have a different setting.

IL:CHATSERVER.PROFILE [Variable]

The value of the variable CHATSERVER.PROFILE appended to the front of IDLE.PROFILE when determining login options etc for the chatserver. The property IDLE.ONLY is also consulted; if T, chatserver only allows connections when machine is in idle mode.

Example:

```
(SETQ CHATSERVER.PROFILE '(ALLOWED.LOGIN (T) IDLE.ONLY T))
```

means to allow only the previously logged in user, and then, only when in IDLE mode.

QUIT [Exec Command]

The QUIT exec command exits a chatserver session. It signals an error if you are not running in a chatserver session.

Documentation for CHATSERVER-NS:

CHATSERVER-NS implements the Xerox Network Systems GAPTELNET protocol. It allows connection to a machine running Medley from other machines that implement this protocol, including Viewpoint (using ViewpointChat), External Communication Service servers (which allow dial-in from remote terminals into an XNS network), XDE workstations and other Interlisp-D implementations (Koto, Lyric, Medley.)

Gaptelnet is a courier server program, and so requires the COURIERSERVE lispusers module (which it loads automatically). The following function is part of COURIERSERVE

(COURIER.START.SERVER) [Function]

This "starts" the courier server process which listens for connection attempts. It is necessary to call this (once) before you can CHAT to your Lisp workstation. (If the process dies for some reason, you will not be able to CHAT until you restart the process.)

Documentation for CHATSERVER-TCP:

This module was an attempt to implement a TCP/TELNET server. Unfortunately, the mechnaism by which it waits for a connection is buggy, and it does not negotiate terminal characteristics properly with the client calling workstation. It is therefore unreliable & may need to be restarted. Using telnet from a Sun to Lisp I've found it was necessary to explicitly tell the Sun not to echo, to send character at a time, etc.

(TCPCHATSERVER) [Function]

It is necessary to call this function to spawn the process that waits for TCP connections.

Documentation for CHATSERVER-RS232:

The CHATSERVER-RS232 module attempts to allow for connections on the RS232 or TTY port on an 1186 or 1108. It uses DLRS232.

(RS232CHATSERVER) [Function]

Spawns a process waiting for a character to be typed on the RS232 port, and then starts a chatserver session.

Other notes:

The server runs a standard (XCL) exec. Note that you can't do graphics; the debugger will not attempt to open a window, only the type-in commands are available, ED will give you the "teletype" editor. Interrupt characters enabled are ^E, ^D, DEL, ^B, ^H and ^T. (Note that currently interrupts are only processed when they are read, and there is no way to interrupt a run-away process.)

Typeout uses a "---more---" style: after (PAGEHEIGHT) lines, the system will prompt you with a "---more---". Type any character, and the more will be erased.

Chatserver assumes you are chatting from a DM2500 emulator, and treats font changes as a switch between bold and regular as appropriate.

The PREEMPTIVE Lispusers module is useful when running chatserver, because it will keep the running process from blocking out the typein process. For some protocol drivers (and the NS server in particular), this is necessary to avoid timeouts.

CHATSERVER advises various facilities in the environment that normally create menus to check to see if the "controlling" keyboard is not the workstation console; these facilities include TTYIN, the editor, the debugger, CHAT. Thus, calls to the editor use the teletype-style editor from Interlisp, while FIX does not generally allow character editing.

CHECKSET

By: >>Your Name<< (>>Your net address<<)

>>Other packages necessary to run this one<<

This document last edited on >>DATE<<

This package checks source files against compiled files in a directory and prompts you about whether you want to (RE)COMPILE the files that need it. It compares the FILECREATED expressions, and determines whether a BRECOMPILE with CHANGES will suffice or if it is necessary to BCOMPL the file.

(CHECKSET FILES COMPFLG) [Function]

FILES is a list of files. If FILES is NIL, CHECKSET is driven by the variable FILESETS.

COMPFLG can be:

N	don't compile, just return list. List can be passed to COMPFILES.
Y or NIL	compile
ASK	ask, for each file, whether to compile it.

FILESETS [Variable]

Used by (CHECKSET NIL): FILESETS is a list of variables, each of which has a value that is a list of files. (CHECKSET NIL) peforms (for X in FILESETS join (CHECKSET (EVALV X))). For example,

if FILESETS = (0LISPSET 1LISPSET) and 0LISPSET = (ATERM LLREAD BREAK), 1LISPSET = (WINDOW EDIT) then (CHECKSET) will check if each of those files in turn need recompiling.

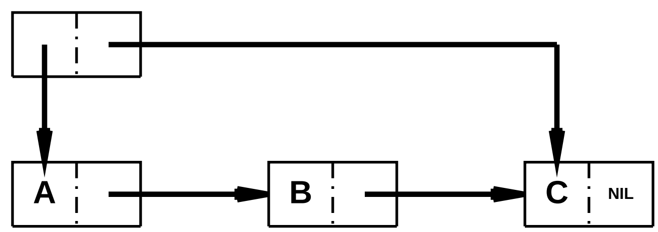
(COMPFILES LST) [Function]

takes a list of elements of the form (RECOMPILE FILE) (COMPILE FILE) as returned by CHECKSET with COMPFLG=N and performs the corresponding operation.

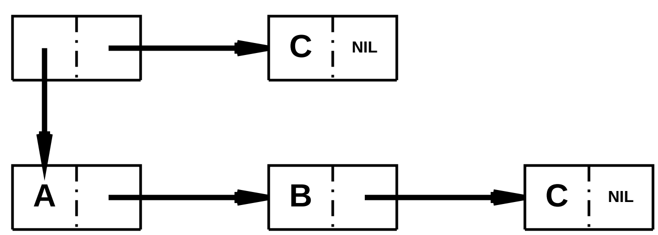
CIRCLPRINT

HPRINT is designed primarily for dumping circular or reentrant list structures (as well as other data structures for which READ is not an inverse of PRINT) so that they can be read back in by Interlisp. The CIRCLPRINT package is designed for printing circular or reentrant structures so that the user can look at them and understand them.

A reentrant list structure is one that contains more than one occurrence of the same EQ structure. For example, TCONC makes use of reentrant list structure so that it does not have to search for the end of the list each time it is called. Thus, if *X* is a list of three elements, (A B C), being constructed by TCONC, the reentrant list structure used by TCONC for this purpose is:

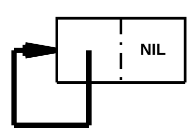


This structure would be printed by PRINT as ((A B C) C). Note that PRINT would produce the same output for the nonreentrant structure:

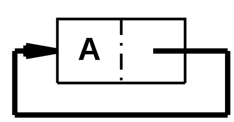


In other words, PRINT does not indicate the fact that portions of the structure in the first figure are identical. Similarly, if PRINT is applied to a circular list structure (a special type of reentrant structure) it will never terminate.

For example, if PRINT is called on the structure:



it will print an endless sequence of left parentheses, and if applied to:



will print a left parenthesis followed by an endless sequence of *A*'s.

The function CIRCLPRINT described below produces output that will exactly describe the structure of any circular or reentrant list structure. This output may be in either single- or double-line format. Below are a few examples of the expressions that CIRCLPRINT would produce to describe the structures discussed above.

First figure, single-line:

((A B *1* C)1)

First figure, double-line:

((A B C) 1)

    1

Third figure, single-line:

(*1* 1)

Third figure, double-line:

(1)

1

Fourth figure, single-line:

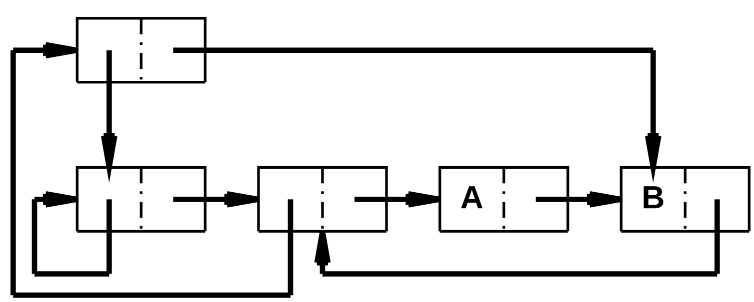
(*1* A . 1)

Fourth figure, double-line:

(A . 1)

1

The more complex structure:



is printed as follows:

Single-line:

(*2* (*1* 1 *3* 2 A *4* B . 3) . 4)

Double-line:

((1   2 A B . 3) . 4)

21   3   4

In both formats, the reentrant nodes in the list structure are labeled by numbers. (A reentrant node is one that has two or more pointers coming into it.) In the single-line format, the label is printed between asterisks at the beginning of the node (list or tail) that it identifies. In the double-line format, the label is printed below the beginning of the node it identifies. An occurrence of a reentrant node that has already been identified is indicated by printing its label in brackets.

(CIRCLPRINT *LIST PRINTFLG RLKNT*) [Function]

Prints an expression describing *LIST*. If *PRINTFLG*=NIL, double-line format is used, otherwise single-line format. CIRCLPRINT first calls CIRCLMARK, and then calls either RLPRIN1 (if *PRINTFLG*=T) or RLPRIN2 (if *PRINTFLG*=NIL). Finally, RLRESTORE is called to restore *LIST* to its unmarked state. Returns *LIST.*

(CIRCLMARK *LIST RLKNT*) [Function]

Marks each reentrant node in *LIST* with a unique number, starting at *RLKNT* plus one (or one, if *RLKNT* is NIL). Value is *RLKNT*. Marking *LIST* physically alters it. However, the marking is performed undoably. In addition, *LIST* can always be restored by specifically calling RLRESTORE.

(RLPRIN1 *LIST*) [Function]

Prints an expression describing *LIST* in the single-line format. Does not restore *LIST* to its unCIRCLMARKed state. *LIST* must previously have been CIRCLMARKed, or an error is generated.

(RLPRIN2 *LIST*) [Function]

Same as RLPRIN1, except that the expression describing *LIST* is printed in the double-line format.

(RLRESTORE *LIST*) [Function]

Physically restores list to its original, unmarked state.

Note that the user can mark and print several structures that together share common substructures, e.g., several property lists, by making several calls to CIRCLMARK, followed by calls to RLPRIN1 or RLPRIN2, and finally to RLRESTORE.

(CIRCLMAKER *LIST*) [Function]

LIST may contain labels and references following the convention used by CIRCLPRINT for printing reentrant structures in single-line format, e.g., (*1* . 1). CIRCLMAKER performs the necessary RPLACAs and RPLACDs to make *LIST* correspond to the indicated structure. Value is (altered) *LIST.*

(CIRCLMAKER1 *LIST)* [Function]

Does the work for CIRCLMAKER. Uses free variables LABELST and REFLST. LABELST is a list of dotted pairs of labels and corresponding nodes. REFLST is a list of nodes containing references to labels not yet seen. CIRCLMAKER operates by initializing LABELST and REFLST to NIL, and then calling CIRCLMAKER1. It generates an error if REFLST is not NIL when CIRCLMAKER1 returns. The user can call CIRCLMAKER1 directly to ‘‘connect up’’ several structures that share common substructures, e.g., several property lists.

CL-TTYEDIT

By: Larry Masinter (Masinter.PA@Xerox.COM)

This document last edited on November 24, 1987.

INTRODUCTION

This file patches the TTY editor so that it is a little more usable in Lyric/Medley for non-Interlisp sources. In particular, it changes the TTY editor so that EDITRDTBL is no longer used; the read table in effect at the time is the ttyeditor is invoked is used instead (*READTABLE*).

It patches the main editor loop (EDITCOM) so that the package and case of edit commands are ignored, i.e., if you type in the P command, it doesn't care whether it is XCL-USER::P or IL:P or |p|.

It patches EDITFPAT (which takes "find" patterns) so that you can specify patterns with --, &, ==, *ANY* in any package, and use --- instead of .. (since symbols consisting entirely of dots are not allowed in CL readtables.)

This file is especially useful if you are talking to another machine using CHATSERVER and need to edit something on the remote machine; since the CHAT connection is character only, you can't run (and the system doesn't attempt to run) SEDIT.

COLORDEMO

Maintained By: Frank Shih (Shih.envos@Xerox.com)

Uses: Color, Peano, ColorPolygons

This document last edited on 8-Nov-88

Color Demonstration Programs

The following functions are on file on COLORDEMO.LCOM.

(COLORDEMO) [Function]

brings up a menu of color demonstration programs. The system cycles through the entries on the menu automatically, allowing each to run for a small, fixed amount of time (typically 40 seconds). Selecting one of the entries in the menu causes it to start that program.

(COLORKINETIC *Wait Window*) [Function]

runs a color version of the standard *Kinetic* demo in *Window.* The demo works by BLTSHADEing random textures to *Window* to random regions of *Window* using random OPERATIONs with a bias towards OPERATION='REPLACE.

(VINEDEMO *Wait Window*) [Function]

draws a twisting vine that changes in thickness, direction, and color as it grows inside *Window*.

(RAINING *Wait Window*) [Function]

drops of rain appear to splash on to *Window* causing concentric circular ripples of color to spread outward on the surface of *Window*.

(MODARTDEMO *Wait Window*) [Function]

some of the art produced by this demo is at least as good as some that you will see in art galleries. The demo actually works by BITBLTing *Window* on to itself with a displacement with random SOURCETYPE and OPERATION, mixed in with some BITBLTed random textures.

(STARBURSTDEMO *Wait Window*) [Function]

far far away in a galaxy somewhere in the future, an unexplained physical force sweeps over peaceful stars turning them at once into brilliant exploding novas which are safely viewed at a distance through the rear view porthole of our fleeing spaceship.

(COLORPEANODEMO *Wait Window*) [Function] the Peano fractal curve in color.

(BUBBLEDEMO *Wait Window*) [Function]

the *Window* fills with brilliantly colored soap bubbles. This demo works by calling FILLCIRCLE.

(OVERPAINTDEMO *Wait*) [Function]

uses masking techniques to print over the lower right color demo window. Notice that just the pixels of the character images get printed and not the white pixels that normally surround the character images.

(TILEDEMO *Wait*) [Function]

takes what currently appears in the four color demo windows and adds their images to a growing list called TILEBITMAPS. The demo then tiles the color screen background followed by repeatedly tileing the four color demo windows with randomly chosen tiles.

(TUNNEL *Speed*) [Function]

draws a series of concentric rectangles of increasing size in increasing color numbers. *Speed* determines the size of the rectangles. This can then be ‘‘run’’ by calling ROTATEIT, which is described below.

(MINESHAFT *N OutFlg*) [Function]

draws a series of concentric rectangles of size *N* in increasing color numbers. *OutFlg* determines whether the color numbers increase or decrease. This can then be ‘‘run’’ by calling ROTATEIT, which is described below.

(WELLDEMO *Wait*) [Function]

draws a series of concentric circles on the color demo windows in increasing color numbers. The circles are then "run" by rotating the color map.

(ROTATEIT *BeginColor EndColor Wait*) [Function]

goes into an infinite loop rotating the screen color map. The colors between *BeginColor* (default zero) and *EndColor* (default maximum color) are rotated. If *Wait* is given, (DISMISS *Wait*) is called each time the color map is changed. This provides an easy way of ‘‘animating’’ screen images.

(COLORPOLYDEMO *ColorStream*) [Function]

is on the file COLORPOLYGONS.DCOM. It runs a version of the *Polygons* program on the color screen.

COLORNNCC

By: >>Your Name<< (>>Your net address<<)

>>Other packages necessary to run this one<<

This document last edited on >>DATE<<

INTRODUCTION

This package is the Xerox Lisp software driver for Number Nine Computer Corporation's Revolution 512 x 8 color card. The IJCAI 1985 show featured an Interlisp-D color window system running on a Xerox 1108 attached to a Busmaster and color display. That color window system was based partly on a version of this package. This advanced high level software has become available with the Lyric release or Xerox Lisp.

NECESSARY HARDWARE

You need a Xerox 1108 with Extended Processor Option (CPE), Xerox Busmaster card, an IBM PC expansion chasis, a Number Nine Computer Corporation Revolution 512 x 8 color card, and a third party color display. Please contact your Xerox representative for details concerning acquiring and setting up all the required hardware.

Assuming you have all the hardware you need, turn it all on. This means

(1) Your 1108 is running Xerox LISP.

(2) Your PC expansion chassis is plugged in and powered on.

(3) A cable connects between your 1108 CPE board and your Busmaster board. (The Busmaster board does not go into the 1108, but should rest outside the 1108.)

(4) Another cable connects between your PC expansion chassis and your Busmaster board.

(5) A pair of purple and orange wires connects your Busmaster board to the +5V/Gnd power supply terminals on the side of your 1108.

(6) Your Number Nine Revolution 512 x 8 board is plugged into the PC expansion chassis.

(7) Your color display is plugged in and powered on.

(8) Three cables for red, green, and blue signal connect your Number Nine card to your color display.

Any reconnections that involve (3), (4), or (5) should be made while your 1108 is off. Until you issue some software commands, a black display is normal.

Check that your hardware is set up correctly by typing

(\COLORNNCC.STARTBOARD) [Function]

Taking less than a second to execute, there should be a noticable flicker on your color display, followed by what can be taken to be a stable abstract pattern representing the contents of the Number Nine card's RAM when the PC chassis was turned on. If \COLORNNCC.STARTBOARD doesn't return and simply waits, there could be something wrong with the BusMaster card. If \COLORNNCC.STARTBOARD does return, but the image is noisy instead of stable, it could be that you have to adjust the frequency selector on your color display.

COLORNNCC SOFTWARE

The COLORNNCC package provides the machine dependent portion of software that is needed to drive your color display assuming you are using an 1108 with COLORNNCC card and Busmaster. Other than LOADing the COLORNNCC package and turning the COLORNNCC package on using the function COLORDISPLAY, all additional functionality is provided by and documented with the COLOR package. There are no COLORNNCC functions that the user needs to call directly. The user calls functions described in the COLOR documentation.

Once your hardware is on, you can proceed to issue COLOR commands to your hardware. You should have the COLORNNCC package already LOADed from your LIBRARY directory. That is, you've already done something like (LOAD '<LIBRARY>COLORNNCC.DCOM). At this point it may be convenient to follow this documentation along with the documentation for COLOR in the Lisp Library Packages Manual. If you now type

```
(COLORDISPLAY 'ON 'REV512X8)
```

your display will now change from total black to a color test pattern with horizontal and vertical stripes. The sequence of events is that there should be a noticable flicker on your color display, followed by what can be taken to be an abstract pattern representing the contents of the Number Nine card's RAM when the PC chassis was turned on, followed by a white wall covering up this abstract pattern, followed by the painting of this white wall with horizontal and vertical strpes of color woven together. There are now some simple tests you can do to satisfy yourself that your hardware is working. Here is a small list of things to try:

```
(SETQ CSBM (COLORSCREENBITMAP))
(BLTSHADE 'WHITE CSBM)
(BLTSHADE 'RED CSBM)
(BLTSHADE 'GREEN CSBM)
(BLTSHADE 'BLUE CSBM)
(SETQ DS (DSPCREATE CSBM))
(DRAWLINE 0 0 500 500 10 'REPLACE DS 'YELLOW)
(DRAWLINE 500 0 0 500 10 'REPLACE DS 'CYAN)
```

Assuming all has gone well to this point, you should now be able to try all the functions described in the COLOR package documentation. The COLORDEMO package is a good source of test programs to try — (IL:LOAD 'COLORDEMO.LCOM) to get this package. Both COLOR and COLORDEMO documentation are in your LispUsers' Manual.

COMMWINDOW

By: Larry Masinter, Stan Lanning, Nick Briggs, Richard Burton (Masinter.pa@Xerox.com, Lanning.PA@Xerox.COM, Briggs.PA@Xerox.com, Burton.pa@Xerox.com)

Uses: COURIERSERVE

This document last edited on 2-Apr-87 17:49:35

INTRODUCTION

This is a demonstration of communication capabilities. The COMMWINDOW module implements a "remote window" capability, where one user can watch (with slow update) a region of another users screen.

Both participants need to have COMMWINDOW loaded.

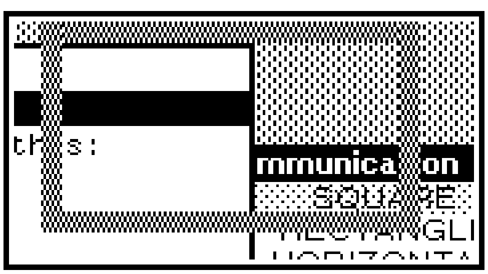
To show someone else a piece of your screen, call

(SEND-BITS *partner &OPTIONAL frame*) [Function]

The partner is the name of the machine you want to talk to. (The watcher has to be registered in the clearinghouse database; this is a NS protocol name.)

frame, if supplied, is a screen region to show. If it is omitted, send-bits will prompt the (sender) for a frame of the screen.

The sender has complete control over the frame. The frame appears as a gray frame around the area shown, like this:



(The frame consists really of 4 thin windows, so that you can really type/button inside it.)

The frame can be moved by left buttoning anywhere on it. Right buttoning on it makes it go away temporarily.

(Reshaping the frame can be accomplished by left buttoning the frame while the shift key is depressed. This doesn’t reshape the remote window currently, so this isn’t very useful.)

Since this is an experiment, there are a couple of parameters you can vary. The function (mode-menu) will bring up a window that lets you control any and all of these parameters.

The first is the shape of the areas sent in each packet.

Currently available shapes are:

square: a square of bits as big as will fit in a single packet

rectangle: 4 times wider than it is high.

horizontal: as wide as the frame, and then as high can be

vertical: as high as the frame and as wide as can be

h3: this is a funny mode, where the update is in horizontal chunks, but spread out.

PARAMETERS

The parameter \ETHERLIGHTNING lets you tell the low-level ethernet code to randomly drop packets. This parameter can be used to study the effects of noisy communication lines on transmision protocol.

Another parameter gives you some local control over the region within the frame that is updated. There are three available values:

Sender: Update near the sender's cursor, when it has moved, ASAP.

Viewer: Update near the viewer's cursor, when it has moved, ASAP.

NIL: don't do anything special.

Another parameter lets lets you prune out sending packets that update regions of the display that have not changed. The tradeoff here between the cost of sending the packets vs testing a region to see if it has changed.

A recent addition was the following part of the protocol: when the mouse on the sender is moving, it will send a square around the mouse interleaved with any other transmission ongoing. That means you can make sure a remote area is being updated by wiggling the mouse.

The sender's mouse is also tracked in the remote viewers viewpoint (the cursor shape is currently not tracked, however). The mouse coordinates are sent every packet, so that mouse position always updates.

The viewer has a limited option to place a pointer back on the sender's screen: if the viewer holds the shift key down while the viewer's mouse is in the view point window, a little pointer () will appear in the senders window and track the viewers cursor. (The pointer is in fact a little icon window.) When the viewer releases the shift key, the pointer icon disappears.

CAUTIONS:

Don't move the frame off the screen. It will just signal an error.

The "don't change unsent tiles" parameter doesn't seem to work.

COMPAREDIRECTORIES

By: Larry Masinter and Ron Kaplan

This document edited on

December 2, 1987

December 28, 1998 (Ron Kaplan)

April 7, 2018 (Ron Kaplan)

Rewritten December, 2021 (Ron Kaplan)

COMPAREDIRECTORIES compares the contents of two directories, identifying files according to their creation dates and lengths. It is called using the function

(COMPAREDIRECTORIES DIR1 DIR2 SELECT INCLUDEDFILES EXCLUDEDFILES USEDIRECTORYDATES OUTPUTFILE ALLVERSIONS) [Function]

Compares the creation dates of files with matching names in the lists that CDFILES returns for DIR1 and DIR2. Collects or prints CDENTRIES for those files that meet the SELECT criteria. May also collect or print entries for relevant files that exist in DIR1 or DIR2 but not both.

SELECT specifies which the match/mismatch criteria for filtering the output. If SELECT is or contains

AFTER or >: select entries where file1 has a later date than file2

BEFORE or <: select entries where file1 has an earlier date than file2

SAMEDATE or =: select entries where file1 and file2 have the same date

-*: exclude entries where file1 does not exist

*-: exclude entries where file2 does not exist

~=: exclude entries where file1 and file2 are byte-equivalent

SELECT = NIL is equivalent to (< > -* *-). Excludes files with matching dates, a useful default for identifying files that may require further attention.

SELECT = T is equivalent to (= < > -* *-). Includes all files for processing by other functions or later filtering by CDSUBSET (below).

SELECT may also contain the token AUTHOR to indicate that authors should be provided in the printed output (see CDPRINT below).

Unless USEDIRECTORYDATES, the FILECREATED date is used for the date comparison of Lisp source and compiled files, otherwise the file-system CREATIONDATE is used.

If OUTPUTFILE=NIL, then the value is a CDVALUE structure with fields (CDPARAMETERS . CDENTRIES). CDPARAMETERS records the parameters in the call to COMPAREDIRECTORIES and CDENTRIES is the list of per-file comparison results. CDPARAMETERS has fields

    CDDIR1 CDDIR2 CDCOMPAREDATE CDSELECT.

CDENTRIES contains one entry for each of the file-comparisons that meets the SELECT criteria. Each entry is a CDENTRY record with fields

    MATCHNAME INFO1 DATERULE INFO2 EQUIV

where MATCHNAME is the name.extension shared by the two files, and each file info is either NIL (for nonexistent files) or a CDINFO record with fields

    (FULLNAME DATE LENGTH AUTHOR TYPE EOL)

TYPE is SOURCE for Lisp source (filecreated) files, COMPILED for Lisp compiled files, otherwise the PRINTFILETYPE (TEXT, TEDIT...) or NIL. EOL is CR, LF, CRLF, or NIL.

When both files exist, the date relation is one of <, =, or >. Otherwise, the date relation is * if only one file exists.

EQUIV is T for files that contain the same bytes. In that case, the date of the earlier file is assumed to be more accurate, it replaces the CREATIONDATE of the earlier file, and the date relation is changed to =.

If OUTPUTFILE is not NIL, then it is a filename or open stream on which selected entries will be printed (T for the terminal) by CDPRINT.

COMPAREDIRECTORIES always sets the variable LASTCDVALUE to the CDVALUE data structure. This is used by the functions below if their CDENTRIES is NIL.

(CDPRINT CDVALUE FILE COLHEADINGS PRINTAUTHOR) [Function]

Prints CDVALUE on FILE, with one line for each entry. The line for each entry is of the form

    FILE1 (AUTHOR) SIZE DATE DATEREL DATE FILE2 (AUTHOR) SIZE

For example

```
ACE.;1 (Joe) 235 2-May-1985 18:03:54 < 30-Sep-1985 11:14:48 ACE.;3 (Sam) 396
```

The line for byte-equivalent files is prefixed with ==. If the files are equivalent except for a difference in end-of-line conventions, the equivalence prefix will indicate the convention for each file (C for CR, L, for LF, 2 for CRLF). Thus C2 indicates that the files are equivalent except that file1 marks line ends with CR and file2 with CRLF.

COLHEADINGS can be a pair (col1 col2) of strings to be printed as column headings.

Note that because COMPAREDIRECTORIES sets LASTCDVALUE, evaluating (CDPRINT) after COMPAREDIRECTORIES prints the results of the last comparision.

For conciseness, authors are included only if PRINTAUTHOR or if AUTHOR is included in the CDSELECT parameter of CDVALUE. Also, redundant file-name hosts/directories are not printed.

(CDTEDIT CDVALUE TITLE COLHEADINGS PRINTAUTHOR) [Function]

Produces the CDPRINT output in a read-only TEDIT window, with TITLE if given.

(CDBROWSER CDVALUE TITLE COLHEADINGS BROWSERPROPS SEPARATEDIRECTIONS MENUITEMS PRINTAUTHOR) [Function]

Produce the CDPRINT output in a TABLEBROWSER window with menu commands for comparing the contents of individual files, viewing files in read-only TEDIT windows, copying files from one directory to another, etc. Lisp source files are compared with COMPARESOURCES, text files with COMPARETEXT. If SEPARATEDIRECTIONS, the entry lines are grouped according to whether the date relation is < or >.

(CDFILES DIR INCLUDEDFILES EXCLUDEDFILES ALLVERSIONS DEPTH) [Function]

Returns a list of full filenames for files in directory DIR (NIL=T=the connected directory) that match the other file-name filtering criteria. Files are excluded if:

Their name does not match a pattern in INCLUDEDFILES (NIL = *.*). Dotted files are excluded unless FILEPATTERNS includes .* and files in subdirectories are excluded if the number of subdirectories exceeds DEPTH (below).

They do not match patterns on the list EXCLUDEDFILES. *.* excludes all extensions, *.COM or just COM excludes extentsions on *COMPILED-EXTENSIONS*. EXCLUDEDFILES contains .* to suppress dotted files unless .* also appears in INCLUDEDFILES.

They are not the highest version unless ALLVERSIONS=T.

DEPTH controls the depth of subdirectory exploration. T means all levels, NIL means no subdirectories. Otherwise the maximum number of > or / characters below the starting DIR in the fullname of files.

(CDFILES) produces all the newest, undotted files in the immediate connected directory.

(CDMERGE CDVALUES) [Function]

Merges all subsets of CDVALUES that have the same CDSELECT into a single CDVALUE with the union of their CDENTRIES. The CDCOMPAREDATE of the merger will be the latest of the dates, and the directories and match names will be adjusted to reflect the original subdirectory sources.

(CDMAP CDVALUE FN) [Function]

(CDSUBSET CDVALUE FN) [Function]

CDMAP and CDSUBSET both apply FN to each CDENTRY in CDVALUE, perhaps modifying the information in the entry. CDSUBSET returns a new cdvalue structure whose entries are the subset of the entries (perhaps modified) for which FN is non-NIL. For convenience, at each invocation the variables MATCHNAME INFO1 DATEREL INFO2 and EQUIV are bound to the corresponding fields and can be used freely by FN.

USEFUL UTILITIES

(FIX-DIRECTORY-DATES FILES) [Function]

For every file included in or specified by FILES, if it is a Lisp source or compiled whose directory creation date is more than 30 seconds later than its internal filecreated date (presumably because of copying), then its directory date is reset to match the internal date. FILES can be a list of file names or a pattern interpretable by FILDIR. Returns a list of files whose dates have been changed.

(FIX-EQUIV-DATES CDVALUE) [Function]

If there is an entry in CDVALUE whose files are EQUIVALENT but with different directory creation dates, the directory date of the file with the later date (presumably a copy) is reset to match the date of the earlier file. In the end all equivalent files will have the same (earliest) date. Returns a list of files whose dates have been changed.

(COPY-MISSING-FILES CDVALUE TARGET MATCHNAMES) [Function]

Target is 1 or 2, indicating the direction of potential copies. If an entry with a source file but no target file has a matchname in MATCHNAMES, the source file is copied to the target directory. All target-absent files are copied if MATCNAMES is NIL. Source properties (including version number) are preserved in the target.

(COPY-COMPARED-FILES CDVALUE TARGET MATCHNAMES) [Function]

TARGET is 1 or 2, indicating the direction of potential copies. If an entry with both source and target files has a matchname in MATCHNAMES, the source file is copied to a new version of the target file. All files are copied if MATCHNAMES is NIL.

(COMPILED-ON-SAME-SOURCE CDVALUE) [Function]

Returns the subset of entries with Lisp compiled files (dfasl or lcom) that are compiled on the same source, according to SOURCE-FOR-COMPILED-P below. Presumably one should be removed to avoid confusion.

(FIND-SOURCE-FILES CFILES SDIRS DFASLMARGIN) [Function]

Returns (CFILE . SFILES) pairs where CFILE is a Lisp compiled file in CFILES and SFILES is list of files in SDIRS that CFILE was compiled on according to SOURCE-FOR-COMPILED-P. This suggests that at least one of SFILES should be copied to CFILE's location (or vice versa).

(FIND-COMPILED-FILES SFILES CDIRS DFASLMARGIN) [Function]

Returns (CFILE . SFILES) pairs where SFILE is a Lisp source file in SFILES and CFILES are files in CDIRS that are compiled on SFILE according to SOURCE-FOR-COMPILED-P. This suggests that at least one of CFILES should be copied to SFILE's location.

(FIND-UNCOMPILED-FILES FILES DFASLMARGIN COMPILEXTS) [Function]

Returns a list of elements each of which corresponds to a source file in FILES for which no appropriate compiled file can be found. An appropriate compiled file is a file in the same location with extension in COMPILEEXTS (defaulting to *COMPILED-EXTENSIONS*) that satisfies SOURCE-FOR-COMPILED-P. Each element is a list of the form

(sourcefile . cfiles)

cfiles contains compiled files that were compiled on a different version of sourcefile, NIL if no such files exist. Each cfile item is a pair (cfile timediff) where timediff is the time difference (in minutes) between the creation date of the compiled-file's source and the creation date of sourcefile (positive if the cfile was compiled later, as should be the case). FILES can be an explicit list of files, or a file specification interpretable by FILDIR; in that case only the newest source-file versions are processed.

(FIND-UNSOURCED-FILES CFILES DFASLMARGIN COMPILEXTS) [Function]

Returns the subset of the compiled files specified by CFILES for which a corresponding source file according to SOURCE-FOR-COMPILED-P cannot be found in the same directory. CFILES can be a list of files or a pattern that FILDIR can interpret. COMPILEEXTS can be one or more explicit compile-file extensions, defaulting to *COMPILED-EXTENSIONS*.

(SOURCE-FOR-COMPILED-P SOURCE COMPILED DFASLMARGIN) [Function]

Returns T if it can confirm that Lisp COMPILED file was compiled on Lisp SOURCE file. SOURCE and COMPILED can be provided as CREATED-AS values, to avoid repetitive computation. This compares the information in the filecreated expressions, original file names and original dates, and not the current directory names and dates.

It appears that the times in DFASL files may differ from the filecreated source dates by a few minutes. The DFASLMARGIN can be provided to loosen up the date matching criterion. DFASLMARGIN is a pair (max min) and a DFASL COMPILED is deemed to be compiled on SOURCE if the compiled's source date is no more than max and no less than min minutes after the source date. A negative min allows for the possibility that the compiled-source date is earlier than the candidate source date. DFASLMARGIN defaults to (20 0). A single positive number x is coerced to (x 0). A single negative number is coerced to (-x x) (compiled file is no more than x minutes later or earlier). T is infinity in either direction. Examples:

(T 0): COMPILED compiled on source later than SOURCE
(0 T): COMPILED compiled on source earlier than SOURCE (odd)
12: COMPILED compiled on source later than SOURCE by no more than 12 minutes -12: COMPILED compiled on source 12 minutes before or after SOURCE

(FIND-MULTICOMPILED-FILES FILES SHOWINFO) [Function]

Returns a list of files in FILES that have more than one type of compiled file (e.g. LCOM and DFASL). FILES is interpretable by FILDIR. If SHOWINFO, then the value contains a list for each file of the form

(rootname loaded-version . CREATED-AS information for each compile-type)

Otherwise just the rootname of the source is returns.

(CREATED-AS FILE) [Function]

If FILE is a Lisp source or compiled file, returns a record of its original filename and filecreated dates, and for compiled files, also the original compiled-on name and date. The return for a source file is a pair

(sfullname sfilecreateddate)

The return for a compiled file is a quadruple

(cfullname cfilecreated sfullname sfilecreateddate)

where sfullname and sourcefilecreated are extracted from the file's compiled-on information. The return is (fullname NIL) for a non-Lisp file.

(EOLTYPE FILE SHOWCONTEXT) [Function]

Returns the EOLTYPE of FILE (CR, LF, CRLF) if the type is unmistakable: contains at least one instance of one type and no instances of any others. Returns NIL if there is evidence of inconsistent types. If SHOWCONTEXT is an integer, it is the number of bytes for EOLTYPE to display before and after an instance of an inconsistent type. At each instance, the user is asked whether to continue scanning for other instances. SHOWCONTEXT = T is interpreted as 100.

(BINCOMP FILE1 FILE2 EOLDIFFOK) [Function]

Returns T if FILE1 and FILE2 are byte-identical. If EOLDIFFOK and FILE1 and FILE2 differ only in their eol conventions, the value is a list of the form (EOL1 EOL2), e.g. (CR CRLF). Otherwise the value is NIL.

COMPARESOURCES

By: Bill van Melle (vanMelle.pa@Xerox.com)

Browser added by Ron Kaplan (12/2021)

INTRODUCTION

COMPARESOURCES is a program for comparing two versions of a Lisp source file for differences. The comparison is completely brute-force: COMPARESOURCES reads the complete contents of both files, and compares all the expressions for differences. The files need not be ones produced by MAKEFILE, as COMPARESOURCES reads the contents with READFILE; however, the program is tuned for files of the type produced by MAKEFILE.

HOW TO USE IT

The interface consists of a two functions, COMPARESOURCES and CSBROWSER.

(COMPARESOURCES *FILEX FILEY EXAMINE DW? LISTSTREAM*) [Function]

Compares the files named *FILEX* and *FILEY* for differences. For each type of file object (function, variable, record, etc), COMPARESOURCES identifies which objects of that type differ, and for each such object prints on *LISTSTREAM* a comparison using the function COMPARELISTS. If an object exists on only one of the two files, this fact is noted instead by the message "*name* is not on *file*".

If *DW?* is true, COMPARESOURCES calls DWIMIFY on each function body before performing the comparison. This is useful for comparing a file made with CLISP prettyprinted with one made without.

If *EXAMINE* is true, COMPARESOURCES calls the editor to allow you to more closely examine expressions that differ. Its value is either T, meaning call the editor in all cases, or an atom or a list of atoms chosen from among the following:

OLD	Call the editor for changed objects that are on both files.
NEW	Call the editor for objects that are on only one file.
MISC	Call the editor for changed but otherwise unclassified expressions.
2WINDOWS	Call the editor separately for each pair of changed objects.

In the OLD and MISC cases, the editor is called on a list of two elements, the two expressions. In the NEW case, the editor is called on just the single new expression.

The value returned by COMPARESOURCES is a list whose elements are of the form (type . names), listing the names by type of all objects found to be different. Expressions of no particular type are identified collectively as "(Other --)".

(CSBROWSER *FILEX FILEY DW? LABEL1 LABEL2*) [Function]

This directs the output of a call to COMPARESOURCESto a scrollable window. Clicking on the output for each identified difference brings up for further examination side-by-side SEDIT windows on the different objects. LABEL1 and LABEL2 are optional strings that override the default way of constructing the title for the broswer window.

FORM OF THE OUTPUT

The output of COMPARESOURCES is in several sections. First, all functions are compared. Then expressions of other types (variables, macros, etc) are compared. When a difference is found, COMPARESOURCES prints the name of the object and calls COMPARELISTS, the same Interlisp function called by COMPARE and COMPAREDEFS. Finally, expressions inside of DECLARE: forms are recursively analyzed in a separate section in the same fashion. All DECLARE: forms of the same applicability (e.g., EVAL@COMPILE DONTCOPY) are handled in the same subsection.

The output of COMPARELISTS takes one of three forms. The usual form is an abbreviated printing of the two expressions with equal elements in the two structures denoted by "&" or "-*n*-" for a subsequence of *n* identical expressions. Identical elements are printed only for purposes of establishing the context of differences. For example,

```
COMPARESOURCES:
(LAMBDA -3- (PROG -16- (for -10- (COND (& (printout & T -4-) -2-)))
(TERPRI --) &))
(LAMBDA -3- (PROG -16- (for -10- (COND (& (printout &   -4-) -2-)))
             &))
```

indicates that in the function `COMPARESOURCES`, an extra argument was added to a `printout` form, and a `(TERPRI --)` expression was added before the final element of the `PROG`. The first 17 elements of the `PROG` form were unchanged, as were the first 11 and last 2 of the `for`.

A more abbreviated form of output occurs when the expressions differ only in a global substitution. In this case, COMPARELISTS prints "(*x* -> *y*)" to denote that all occurrences of *x* in the first expression were replaced by *y* in the second expression, and there were no other changes.

Finally, COMPARELISTS prints "`SAME`" if the expressions are "the same". Since COMPARESOURCES only calls COMPARELISTS when the two expressions are not EQUAL, the output `SAME` specifically means that the expressions differ only in the bodies of comments (which COMPARELISTS ignores).

USER EXTENSIONS

COMPARESOURCES already "knows" about several kinds of file package objects, including FNS, VARS, MACROS, RECORDS, and PROPS. Any expression not identifiable as some particular type is compared as a vanilla expression. You can extend the set of types it knows about by adding to the following list:

COMPARESOURCETYPES [Variable]

The elements of this list are lists of the form

```
(TYPE  PREDICATEFN  COMPAREFN  IDFN  DESCRIPTION)
```

as follows:

TYPE — The file package type of the object (or whatever name you wish to give it in the case of fictitious object types).

PREDICATEFN A function of one argument, a single top-level expression as read from the file, that returns true if the expression is of the desired type.

COMPAREFN A function of three arguments, one expression from each file (both guaranteed to have satisfied the PREDICATEFN), and the listing stream. COMPAREFN should compare the two expressions in some appropriate way, printing its results to the listing stream. A typical COMPAREFN calls the function COMPARELISTS on some subform of the expressions. If COMPAREFN is NIL, COMPARELISTS is used.

IDFN A function of one argument, an expression, that returns the "name" of the object described by the expression. Two expressions are assumed to define the same object if their names are EQUAL. The name corresponds roughly to a file package name. For example, for type VARS it is the variable name; for type PROPS it is a pair (atom propname). If IDFN is NIL, CADR is used.

DESCRIPTION A string identifying the kind of object, for use in the comparison printout. If DESCRIPTION is NIL, (L-CASE TYPE T) is used.

COMPARETEXT

By Mike Sannella. Tested in Medley by Larry Masinter, updated by Ron Kaplan 12/2021

Uses TEDIT, GRAPHER, REGIONMANAGER

INTRODUCTION

COMPARETEXT is a rather non-standard text file comparison program which tries to address two problems: (1) the problem of detecting certain types of changes, such as detecting when a paragraph is moved to a different part of a document; and (2) the problem of showing the user what changes have been made in a document.

The text comparison algorithm is an adaptation of the one described in the article "A Technique for Isolating Differences Between Files" by Paul Heckel, in CACM, V21, #4, April 1978. The main idea is to break each of the two text files into "chunks" (words, lines, paragraphs, ...), hash each chunk into a hash value, and match up chunks with the same hash value in the two files. This method detects switching two chunks, or moving a chunk anywhere else in the document.

COMPARING TEXT FILES

Two text files can be compared with the following function:

(COMPARETEXT *FILE1 FILE2 HASH.TYPE CHUNKREGION FILELABELS TITLE TEXTWIDTH TEXTHEIGHT*) [Function]

FILE1 and *FILE2* are the names of the two files to compare. The order is not important, except that in the resulting graph the *FILE1* information will appear on the left, and the *FILE2* info on the right. The files may also be provide as input streams.

HASH.TYPE determines how "chunks" of text are defined; how fine-grained the comparison will be. This can be PARA to hash by paragraphs (delimited by two consecutive CRs), LINE to hash by lines (delimited by one CR), or WORD to hash words (delimited by any white space). *HASH.TYPE*=NIL defaults to PARA.

CHUNKREGION is the region on the display screen used for the file comparison graph, the chunk window. If *CHUNKREGION*=NIL, the system asks the user to specify a region, prompting with a region that is just wide enough for the graph. If *CHUNKREGION*=T, a region in the lower left corner is used. IF *CHUNKREGION* is a position, the chunkwindow will be located relative to that position, with its horizontal midpoint at the specified XCOORD and its top at the YCOORD.

FILELABELS is an optional pair of labels that will appear over the columns of the difference graph instead of the (often overly long) full names of the files.

TITLE is an optional title to be used for the comparison window.

TEXTWIDTH and *TEXTHEIGHT* are optional parameters that control the size of each of the two text-display windows.

COMPARETEXT creates a graph with two columns. Each column contains the label for one of the files, and lists the chunks from that file. Each chunk is represented by an atom NNN:MMM, where NNN is the file pointer of the beginning of the chunk within the file, and MMM is the length of the chunk. Lines are drawn from one column to the other to show which chunks in one file are the same as those in the other file. Chunks with no lines going to them do not exist in the other file. [Note: a series of chunks in one file which are the same as a series of chunks in the other file are merged into one big chunk. A series of unconnected chunks is also merged.]

Pressing the LEFT mouse button over one of the chunk nodes causes the node and its counterpart in the other column to be inverted, and read-only Tedit windows are open on the files with the appropriate text selected. If a Tedit window to a file is already active, the selection is simply moved. If COMPARETEXT.AUTOTEDIT is true (initially), then regions are selected automatically for the Tedit windows, otherwise the mouse must be used to specify ghost regions.

Pressing the MIDDLE mouse button over a chunk node raises a pop-up menu with the items: PARA, LINE, and WORD. If one of these is selected, COMPARETEXT is called to compare the selected chunk with the last selected chunks (the ones that are boxed), using the hash type selected, and create a new graph window.

White space (space, tab, CR, LF) is used to delimit chunks, but is ignored when computing the hash value of a chunk. Therefore, if two paragraphs are identical except that one has a few extra CRs after it, they will be considered identical by COMPARETEXT.

If the variable COMPARETEXT.ALLCHUNKS is NIL (initially T), then the graph is abbreviated so that nodes for identical chunks in the same position are not shown.

COMPILEBANG

By: Nick Briggs (Briggs.pa@Xerox.com)

Required by TRILLIUM

This provides an interface to the compiler that avoids the interview for the common cases of in-core compilation. It contains a single function COMPILE!, and the Lispx and edit macros C:

(COMPILE! *X NOSAVE NOREDEFINE PRINTLAP*) [Function]

Calls the compiler to compile *X*. If *X* is a litatom, its definition is compiled and stored in the function cell unless *NOREDEFINE*, and the old definition if any is saved on the property list unless *NOSAVE*. No printing of lap or machine code is done unless *PRINTLAP*.

Thus, to simply compile the function BAR, do COMPILE!(BAR).

X may also be a list form. In this case, COMPILE! assumes that the user is interested just in seeing how that form would compile. The form is embedded in a Lambda expression and compiled. Of course, there is no function-cell to be stored into or saved.

C [Lispx Macro]

The LISPXMACRO C calls COMPILE!, with *PRINTLAP* on, on the next element of the input line. Thus, C BAR will compile, redefine, and save the old definition for BAR.

C (CONS) will show how a call to CONS would compile.

The editmacro C calls COMPILE! on the current expression if it is a list, or on the form of which the current expression is an element.

COURIERDEFS

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

COURIERDEFS contains a procedure-less Courier program, called INTERLISP, which defines several Envos Lisp types as Courier constructed types or as new Courier primitive types (via a COURIERDEF property) for use with Courier server and client programs. The defined Envos Lisp types include:

ATOM Converts to a string on writing and converts to an atom on reading.

FONT Converts a FONTDESCRIPTOR to a record describing the font on writing and convert the record back to a FONTDESCRIPTOR on reading.

REGION A sequence of INTEGER.

POSITION Converts a POSITION record to two integers on writing and converts back to a POSITION record on reading.

NUMBER Like INTEGER but can also be NIL.

BRUSH Converts the various possibilities for a brush (NIL, INTEGER, BRUSH RECORD etc.) to a CHOICE record on writing, converts back to original specification on reading.

OPERATION An ENUMERATION of NIL, REPLACE, PAINT, INVERT or ERASE.

TEXTURE Converts a TEXTURE, NIL or T to a CARDINAL on writing, returns a CARDINAL on reading.

This file is loaded by several other modules that define Courier servers and clients. A Courier program can use the types defined in the INTERLISP program by using the INHERITS slot in the Courier program definition.

COURIEREVALSERVE

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: COURIERSERVE

COURIEREVALSERVE implements both the client and server routines for the simple remote evaluation server described in the COURIERSERVE documentation.

The module defines two user functions:

(REMOTEEVAL *FORM COURIERSTREAM [NOERRORFLG]*) [Function]

(REMOTEAPPLY *FN ARGS COURIERSTREAM [NOERRORFLG]*) [Function]

COURIERSTREAM is obtained by calling COURIER.OPEN to connect with a host that is running the Courier server and has COURIEREVALSERVE loaded. If the *NOERRORFLG* is non-NIL, it is returned if an error is signaled by the remote host, otherwise the functions generate an error.

Due to the removal of ERRORN as of the Lyric release, the error handling is not as informative as in earlier versions. If you are connected to a pre-Lyric host, errors will work as before, otherwise instead of signaling the actual remote error (eg. “*Undefined car of form*”) the generic “*Remote evaluation error!*” error is raised. This is to maintain backward compatibility in the EVAL Courier program. Hopefully, this will be replaced by a new version of the EVAL program designed to correctly remote the new condition-based error handler.

COURIERIMAGESTREAM

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: COURIERSERVE, COURIERDEFS and BITMAPFNS

COURIERIMAGESTREAM implements a Courier client and server program which allows remote hosts to do *image stream* manipulations on other workstations via the network. To do this, it defines the COURIER virtual image stream type which allows the user to manipulate remote image streams through local image streams.

THE IMAGESTREAM COURIER PROGRAM

The module defines a Courier program called IMAGESTREAM (which inherits from the INTERLISP Courier program defined in COURIERDEFS). For each IMAGEOP in the IMAGEOPs definition, there is an equivalent Courier procedure in the IMAGESTREAM program. The module contains the code for both the Courier client and server.

OPENING AND CLOSING REMOTE IMAGE STREAMS

Remote image streams can be opened using either the COURIER image stream type or using direct Courier calls.

The COURIER Image Stream Interface

Remote Courier image streams can be opened using:

```
(SETQ COURIERSTREAM (COURIER.OPEN HOST)
(OPENIMAGESTREAM COURIERSTREAM 'COURIER OPTIONS)
```

which returns an image stream. The OPTIONS can include FILE and IMAGETYPE which are passed to OPENIMAGESTREAM on the remote host and if not supplied, a nameless DISPLAY image stream is opened. All other options are passed to the remote image stream. The image stream can be closed using CLOSEF.

The Courier Procedure Call Interface

Courier image streams can also be opened using Courier procedure calls from any Courier client with the Courier procedure:

```
(OPEN 0 (FILE IMAGETYPE) RETURNS (HANDLE) REPORTS NIL)
```

which is invoked from Lisp by doing:

```
(COURIER.CALL COURIERSTREAM 'IMAGESTREAM 'OPEN FILE IMAGETYPE OPTIONS)
```

where FILE, IMAGETYPE and OPTIONS are similar to the arguments to OPENIMAGESTREAM.

This call will return a handle to be used with the remainder of the IMAGESTREAM procedures. To close an image stream from a Courier client use the Courier procedure:

```
(CLOSE 1 (HANDLE) RETURNS NIL REPORTS NIL)
```

which is invoked from Lisp by doing:

```
(COURIER.CALL COURIERSTREAM 'IMAGESTREAM 'CLOSE HANDLE)
```

DIFFERENCES BETWEEN IMAGEOPS AND IMAGESTREAM COURIER PROCEDURES

All of the IMAGEOPs are implemented in the COURIER image stream type as it merely passes the call to the IMAGEOPs of another image stream type on the remote host. No error checking is done, so invoking an illegal IMAGEOP will cause a Courier rejection of the call.

The arguments to the IMAGESTREAM Courier procedures are generally in the same order as the arguments to the various IM* functions which implement an image stream (stream argument first). An exception is BITBLT (and SCALEDBITBLT) which is defined as follows:

```
(BITBLT 32 (HANDLE BULK.DATA.SOURCE LEFT BOTTOM WIDTH HEIGHT
            SOURCETYPE OPERATION TEXTURE CLIPPINGREGION)
```

The BULK.DATA.SOURCE argument is used to transfer the bitmap using WRITEBINARYBITMAP. This is only relevant to direct Courier calls, the COURIER image stream BITBLT operation hides the differences.

When using the COURIER image stream type, the STRINGWIDTH, CHARWIDTH etc. IMAGEOPs are handled locally, not via Courier calls, to improve efficiency.

IMAGESTREAM PROGRAM VERSIONS

The current implementation of the IMAGESTREAM Courier program is version **1**. This module also has the previous version of the program (**0**) defined as OLDIMAGESTREAM (just the procedure definitions that differ, it inherits from IMAGESTREAM). This allows the current version of the program to accept calls from older versions, but not vice-versa. However, the new version of the IMAGESTREAM Courier program can be loaded and used with the old (pre-Lyric) functions.

COURIERSERVE

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

COURIERSERVE implements a Courier server process for Envos Lisp allowing other hosts to make Courier calls into the workstation. The server supports both multiple Courier stream connections as well as *expedited* (single packet) and *broadcast* calls.

STARTING A COURIER SERVER

The Courier server can be started by evaluating:

(COURIER.START.SERVER *[RESTART]*) [Function]

Once the server is running, it can be invoked by a remote host client using COURIER.OPEN for a Courier stream connection or by using COURIER.EXPEDITED.CALL or COURIER.BROADCAST.CALL for expedited calls. The functions for making Courier client calls from Lisp are documented in the *Interlisp-D Reference Manual* (pages 31.15–31.26).

(COURIER.RESET.SOCKET) [Function]

(Re)Opens and closes the Courier socket. Not normally a user routine, this function is called by COURIER.START.SERVER but it can be called directly if "*socket already open!*" errors persist on the Courier socket (5).

DEFINING A COURIER SERVER FUNCTION

Defining a Courier server program is identical to defining a client program except for the additional field IMPLEMENTEDBY in each procedure in the PROCEDURES section of the Courier program definition:

```
PROCEDURES
((LAYOUT 0 (GRAPHNODES ROOTIDS FORMAT FONT MOTHERD PERSONALD FAMILYD)
            RETURNS (GRAPH)
            REPORTS (LAYOUT.ERROR)
            IMPLEMENTEDBY GRAPH.REMOTELAYOUT))
```

The order of the RETURNS, REPORTS and IMPLEMENTEDBY fields is significant and should be maintained.

The *server function*, named in the IMPLEMENTEDBY field, is invoked when a Courier call to the procedure is made. The server function is applied to the Courier stream, the Courier program and the Courier procedure followed by the arguments named in the Courier definition. The arguments for GRAPH.REMOTELAYOUT would be (COURIERSTREAM PROGRAM PROCEDURE GRAPHNODES ROOTIDS FORMAT ...).

Note that the COURIERSTREAM, PROGRAM and PROCEDURE arguments are not necessarily used, they are made available for implementing special servers.

RETURNING VALUES FROM A COURIER PROGRAM

Results or errors can be returned by a Courier server function by one of two different methods. In the usual, simple case, the server function can return as its result a list starting with one of RETURN, ABORT or REJECT followed by the appropriate values.

For the RETURN result, the tail of the list should be the results as defined in the Courier procedure definition, eg. `(RETURN 23 "John")`.

For the ABORT result, the tail of the list should contain the reason for the abnormal termination (as defined in the Courier program), followed by any error arguments, eg. `(ABORT NAME.NOT.FOUND "John")`.

For the REJECT result, the tail of the list should contain the rejection error as defined in the Courier standard. The only rejection that should occur inside a server function should be UNSPECIFIED if the program needs to reject for any reason. The other rejection types are handled by the Courier server.

Alternatively, the server function can return results directly to the Courier stream and return NIL as its result. To return results directly to the Courier stream use:

(COURIER.RETURN *COURIERSTREAM PROGRAM PROCEDURE RESULTLST*) [Function]

(COURIER.ABORT *COURIERSTREAM PROGRAM ERROR RESULTLST*) [Function]

(COURIER.REJECT *COURIERSTREAM ERROR RESULTLST*) [Function]

EXPEDITED AND BROADCAST COURIER CALLS

The Courier server allows expedited and broadcast Courier calls. The only difference the server function would see if invoked due to an expedited call is that the Courier stream it is handed is actually a record containing an XIP packet and a socket. If the server function does not use the Courier stream directly, then this difference is invisible.

If the server function actually needs a Courier stream to operate (eg. an NS CHAT server), then it should probably include an USE.COURIER abort error in its definition. If the server function needs a Courier stream due to *bulk data* arguments, this will be trapped in the Courier server itself, which will reject appropriately and not invoke the server function.

USING BULK DATA IN A SERVER FUNCTION

If a server function takes a bulk data argument (either BULK.DATA.SINK or BULK.DATA.SOURCE), it is handed an open bulk data stream for that argument when invoked. If the server function returns a result by returning one of the RETURN or ABORT forms as its result, the bulk data stream will be closed automatically. If the server function returns results directly to the Courier stream using COURIER.RETURN or COURIER.ABORT, then the server function must first close the bulk data stream using:

(CLOSE.BULK.DATA *STREAM [ABORTFLG]*) [Function]

The CLOSEF function does not work on the bulk data stream argument and using it will hang the Courier connection. Only the *immediate* bulk data transfer type is handled. NULL, ACTIVE or PASSIVE bulk data transfer types will cause a Courier rejection of type UNSPECIFIED.

SIMPLE SERVER DEFINITION

Below is the Courier definition for a simple evaluation server. The two functions EVAL.REMOTE and APPLY.REMOTE are all that would need to be defined to make the server run:

```
((1105 0)
 TYPES       ((SEXPR STRING)
              (FN STRING)
              (ARGS (SEQUENCE SEXPR))
              (ERRORN (RECORD (ERROR.NUMBER CARDINAL)
                              (ERROR.MESSAGE SEXPR))))
 PROCEDURES ((EVAL 0 (SEXPR)
                   RETURNS (SEXPR)
                   REPORTS (REMOTE.EVAL.ERROR REMOTE.READ.ERROR)
                   IMPLEMENTEDBY EVAL.REMOTE)
              (APPLY 1 (FN ARGS)
                   RETURNS (SEXPR)
                   REPORTS (REMOTE.APPLY.ERROR REMOTE.READ.ERROR)
                   IMPLEMENTEDBY APPLY.REMOTE))
 ERRORS      ((REMOTE.EVAL.ERROR 0 (ERRORN))
              (REMOTE.APPLY.ERROR 1 (ERRORN))
              (REMOTE.READ.ERROR 2 (ERRORN)))
)
```

RELATED FILES

The modules CHATSERVER-NS, COURIERDEFS, COURIEREVALSERVE, COURIERIMAGESTREAM, MONITOR, REMOTEPSW and NSTALK all define Courier servers and/or Courier type definitions.

CROCK

By: Kelly Roach

New Owner: Herb Jellinek (Jellinek.pa@Xerox.com)

CROCK sets up an analog face clock in the user's environment. To use, LOAD CROCK.LCOM and call (CROCK). CROCK requires that PROCESSWORLD be running (automatic in Fugue or later).

CROCK

Function CROCK has the form

(CROCK *REGION*) [Function]

The first invocation creates a clock window, CROCKWINDOW, occupying REGION with style CROCK.DEFAULT.STYLE. If REGION is left NIL, a region will be prompted for. Subsequent invocations use CROCKWINDOW. Only one clock window may exist at any given time. The clock is updated once a minute.

STYLE

The clock's style is maintained as a property list and can be found by (WINDOWPROP CROCKWINDOW 'STYLE). There are four independent boolean properties which the user may control: HANDS (the hands of the clock), TIMES (time digits printed where the hands end), RINGS (rings on the clock face), and NUMBERS (12 numbers around the outside of the clock face). The style first used will be CROCK.DEFAULT.STYLE (bound to '(HANDS T TIMES NIL RINGS NIL NUMBERS T) when CROCK is first loaded).

CROCK.DATEFORMAT

The user can control how the date will be printed in CROCKWINDOW. CROCK.DATEFORMAT should have the form (DATEFORMAT . <tokens>) where each <token> is one of NO.DATE, NO.TIME, NUMBER.OF.MONTH, YEAR.LONG, SLASHES, SPACES, NO.LEADING.SPACES, TIME.ZONE, or NO.SECONDS. These are all listed on pp23.57-58 of the IRM. Unfortunately, some other possibilities, such as DAY.OF.WEEK have not been implemented by Interlisp-D yet and are therefore not available to CROCK yet. The default value for CROCK.DATEFORMAT is (DATEFORMAT NO.SECONDS). For example,

(SETQ CROCK.DATEFORMAT

'(*DATEFORMAT SLASHES NUMBER.OF.MONTH NO.SECONDS*))

would make CROCK print a date string like

28/09/84 14:53

instead of a date string like

28-Sep-84 14:53

Since CROCK updates itself only once a minute, it is probably a good idea to always include NO.SECONDS in your CROCK.DATEFORMAT.

CROCK.ALARM AND CROCK.TUNE

The user can set CROCK's alarm via

(CROCK.ALARM *DATESTRING*) [Function]

where DATESTRING is any arg acceptable to Interlisp's IDATE (such as the date CROCK prints in CROCKWINDOW). CROCK will act appropriately when time reaches DATESTRING. Dandelion users can set global CROCK.TUNE to a tune to be played by Interlisp's PLAYTUNE when CROCK's alarm acts.

RECOMMENDED USAGE

The simplest way to call CROCK from your init file or other function is to set your CROCK globals, then call CROCK:

(SETQ CROCK.DEFAULT.STYLE *STYLE*) [Variable]

(SETQ CROCK.DATEFORMAT *DATEFORMAT*) [Variable]

(SETQ CROCK.TUNE *TUNE*) [Variable]

(CROCK *REGION*) [Function]

You supply <style>, <dateformat>, <tune>, and <region>. You only need the SETQs if you want non-default values. If no <region> is supplied, CROCK will prompt for one.

LEFT MOUSE BUTTON

Buttoning CROCKWINDOW with the left mouse button requests immediate update of the clock. (Of course, it may take a while for the process scheduler to get to it.)

MIDDLE MOUSE BUTTON

Buttoning CROCKWINDOW with the middle mouse button presents a menu of commands for modifying the clock's style. Menu item SHOW.STYLE prints the clock's style.

RIGHT MOUSE BUTTON

Buttoning CROCKWINDOW with the left mouse button presents the usual window menu. RESHAPEing the CROCKWINDOW causes the clock to change its size to fit the new window region. CLOSEing the CROCKWINDOW deletes the clock process.

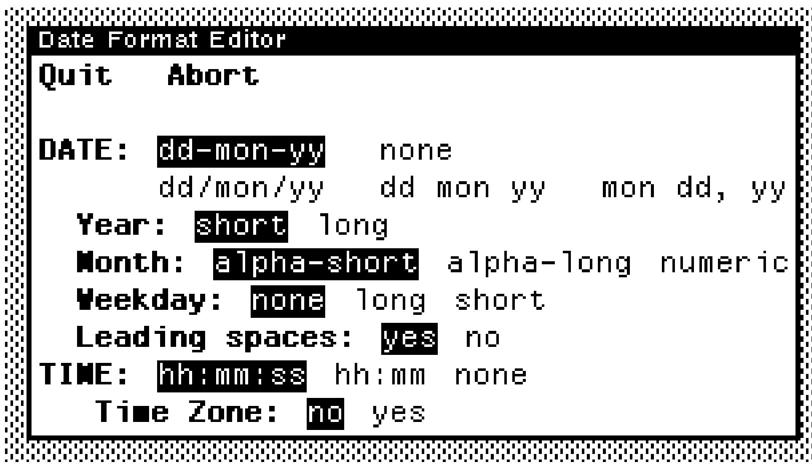
DATEFORMAT-EDITOR

By: Johannes A. G. M. Koomen
(Koomen.wbst@Xerox.com or Koomen@CS.Rochester.edu)

This document last edited on May 19, 1989 by Bill van Melle.

DESCRIPTION

DATEFORMAT-EDITOR provides a menu-based interface for creating and editing date formatting lists (see IRM, Section 12.5). The menu is a Free Menu (see FREEMENU in Medley Release Notes), and looks like:



INTERFACE

(EDIT-DATEFORMAT *DATEFORMAT*) [Function]

DATEFORMAT is either NIL or the value returned from a call to the function DATEFORMAT (see IRM, Section 12.5). EDIT-DATEFORMAT starts by pre-selecting date formatting keys according to DATEFORMAT, or default ones if DATEFORMAT is NIL. It then enters a busy-wait loop, blocking until the DateFormat Editor window is closed, or **Quit** or **Abort** is selected. EDIT-DATEFORMAT returns a new value obtained from the function DATEFORMAT given the selected date formatting keys if **Quit** was selected, otherwise NIL.

DATEFORMAT-EDITOR-ITEMS [Variable]

A list of items acceptable to the function FM.FORMATMENU (see FREEMENU in the Release Notes). Unfortunately, some of the date format details are embedded in the DateFormat Editor, rather than in these items, so leave ID's and LABEL's alone, otherwise mung around to your heart's content if you desire a different layout for the DateFormat Editor. Initial value is reflected by the screen snap above.

(GET-DATEFORMAT-EDITOR *RECOMPUTE?*) [Document Object]

Returns the FreeMenu window of the DateFormat Editor. If RECOMPUTE? is non-NIL, recomputes the FreeMenu. Use this funciton with argument T if you change the variable DATEFORMAT-EDITOR-ITEMS.

EXTENDED DATEFORMAT OPTIONS

DATEFORMAT-EDITOR supports the following additional DATEFORMAT options by virtue of loading the module DATEPATCH:

MONTH.LONG [DateFormat Option]

Provides for full names of months rather than the first three characters. For instance, "20 February 1987" is produced by (GDATE NIL (DATEFORMAT MONTH.LONG YEAR.LONG SPACES NO.TIME)) .

MONTH.LEADING [DateFormat Option]

Causes the month to appear before the day. For instance, "February 20, 1987" is produced by (GDATE NIL (DATEFORMAT MONTH.LEADING MONTH.LONG YEAR.LONG NO.TIME)). MONTH.LEADING implies SPACES and disables NUMBER.OF.MONTH.

DATESORT
[Internal Lafite Utility]

Datesort places an item on the background menu. When selected you will be prompted for the name of a Lafite Mail folder to be sorted, and the name of a new resulting sorted folder. The folder messages will then be sorted in ascending order on their "Date:" fields (this takes some time).

DEBUGGER-CONTEXT

By: Herb Jellinek (Jellinek.pa@xerox.com)

This document last edited on August 13, 1987.

Introduction

When debugging Common Lisp programs, have you ever wished that the Xerox Lisp debugger let you do things in the right lexical context? Ever wish you could access locally-defined functions, return from blocks, or evaluate variables in the debugger without resorting to the inspector? Ever wish you could tell the "big boys" in Washington a thing or two? Well, two out of three ain't bad. DEBUGGER-CONTEXT gives you the ability to access and modify the lexical context of code you are debugging in a straightforward way.

User Interface

DEBUGGER-CONTEXT defines two debugger commands that affect the debugger's lexical context.

`lex` [Debugger Command]

Sets the lexical context of the debugger window to that of the stack frame selected in the backtrace window (`il:lastpos`). Henceforth, all evaluation done in this debugger window will be with respect to that lexical environment. The lexical environment of a frame is taken to be the first one encountered as an argument in that frame. Once the `lex` command is given, selecting other stack frames in the backtrace window will not change the lexical environment; you must issue another `lex` command to do so.

The lexical environment of a frame is taken to be the first one encountered as an argument in that frame.

`unlex` [Debugger Command]

Removes the current lexical context; all evaluation will subsequently be done with respect to the current dynamic environment.

DECL

UNSUPPORTED

INTERNAL

Uses: SIMPLIFY, LABEL and LAMBDATRAN

NOTE TO LYRIC/MEDLEY USERS

The DECL module is not supported in Lyric/Medley since it uses the DWIM facilities heavily, and DWIM is not supported in Lyric/Medley. It is being released as a LispUsers module only for backward compatibility. The DECL module only runs under the OLD-INTERLISP-T executive, and all the code that uses it will also have to run under this executive. Therefore, you may wish to convert code that uses DECL to something else.

INTRODUCTION

The Decl LispUsers package is contained on the file DECL.LCOM. The Decl package requires the LambdaTran package. LAMBDATRAN.LCOM will automatically be loaded with Decl if it is not already present.

The Decl package extends Interlisp to allow the user to declare the types of variables and expressions appearing in functions. It provides a convenient way of constraining the behavior of programs when the generality and flexibility of ordinary Interlisp is either unnecessary, confusing, or inefficient.

Decl provides a simple language for declarations, and augments the interpreter and the compiler to guarantee that these declarations are always satisfied. The declarations make programs more readable by indicating the type, and therefore something about the intended usage, of variables and expressions in the code. They facilitate debugging by localizing errors that manifest themselves as type incompatibilities. Finally, the declaration information is available for other purposes: compiler macros can consult the declarations to produce more efficient code; coercions for arguments at user interfaces can be automatically generated; and the declarations will be noticed by the Masterscope function analyzer.

The declarations interpreted by the Decl package are in terms of a set of declaration types called *decltypes*, each of which specifies a set of acceptable values and also (optionally) other type-specific behavior. The Decl package provides a set of facilities for defining decltypes and their relations to each other, including type-valued expressions and a comprehensive treatment of union types.

The following description of the Decl package is divided into three parts. First, the syntactic extensions that permit the concise attachment of declarations to program elements are discussed. Second, the mechanisms by which new decltypes can be defined and manipulated are covered. Finally, some additional capabilities based on the availability of declarations are outlined.

USING DECLARATIONS IN PROGRAMS

Declarations may be attached to the values of arbitrary expressions and to LAMBDA and PROG variables throughout (or for part of) their lexical scope. The declarations are attached using constructs that resemble the ordinary Interlisp LAMBDA, PROG, and PROGN, but which also permit the expression of declarations. The following examples illustrate the use of declarations in programs.

Consider the following definition for the factorial function (FACT *N*):

```
[LAMBDA (N)
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

Obviously, this function presupposes that *N* is a number, and the run-time checks in ITIMES and SUB1 will cause an error if this is not so. For instance, (FACT T) will cause an error and print the message NON-NUMERIC ARG T. By defining FACT as a DLAMBDA, the Decl package analog of LAMBDA, this presupposition can be stated directly in the code:

```
[DLAMBDA ((N NUMBERP))
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

With this definition, (FACT T) will result in a NON-NUMERIC ARG T error when the body of the code is executed. Instead, the NUMBERP declaration will be checked when the function is first entered, and a *declaration fault* will occur. Thus, the message that the user will see will not dwell on the offending value T, but instead give a symbolic indication of what variable and declaration were violated, as follows:

```
DECLARATION NOT SATISFIED
((N NUMBERP) BROKEN):
```

The user is left in a break from which the values of variables, e.g., *N*, can be examined to determine what the problem is.

The function FACT also makes other presuppositions concerning its argument, *N*. For example, FACT will go into an infinite recursive loop if *N* is a number less than zero. Although the user could program an explicit check for this unexpected situation, such coding is tedious and tends to obscure the underlying algorithm. Instead, the requirement that *N* not be negative can be succinctly stated by declaring it to be a subtype of NUMBERP that is restricted to non-negative numbers. This can be done by adding a SATISFIES clause to *N*'s type specification:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N])
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

The predicate in the SATISFIES clause will be evaluated after *N* is bound and found to satisfy NUMBERP, but before the function body is executed. In the event of a declaration fault, the SATISFIES condition will be included in the error message. For example, (FACT -1) would result in:

```
DECLARATION NOT SATISFIED
```

```
((N NUMBERP (SATISFIES (NOT (MINUSP N))) BROKEN):
```

The DLAMBDA construct also permits the type of the value that is returned by the function to be declared by means of the pseudo-variable RETURNS. For example, the following definition specifies that FACT is to return a positive integer:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N]
   [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
  (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

After the function body is evaluated, its value is bound to the variable VALUE and the RETURNS declaration is checked. A declaration fault will occur if the value is not satisfactory. This prevents a bad value from propagating to the caller of FACT, perhaps causing an error far away from the source of the difficulty.

Declaring a variable causes its value to be checked not only when it is first bound, but also whenever that variable is reset by SETQ within the DLAMBDA. In other words, the type-checking machinery will not allow a declared variable to take on an improper value. An iterative version of the factorial function illustrates this feature in the context of a DPROG, the analog of PROG:

```
(DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N]
   [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
[DPROG ([TEMP 1 FIXP (SATISFIES (IGREATERP TEMP 0]
   [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
      LP  (COND ((EQ N 0) (RETURN TEMP)))
          (SETQ TEMP (ITIMES N TEMP))
          (SETQ N (SUB1 N))
          (GO LP]
```

DPROG declarations are much like DLAMBDA declarations, except that they also allow an initial value for the variable to be specified. In the above example, TEMP is declared to be a positive integer throughout the computation and N is declared to be non-negative. Thus, a bug which caused an incorrect value to be assigned by one of the SETQ expressions would cause a declaration failure. Note that the RETURNS declaration for a DPROG is also useful in detecting the common bug of omitting an explicit RETURN.

DLAMBDAs

The Decl package version of a LAMBDA expression is an expression beginning with the atom DLAMBDA. Such an expression is a function object that may be used in any context where a LAMBDA expression may be used. It resembles a LAMBDA expression except that it permits declaration expressions in its argument list, as illustrated in the examples given earlier. Each element of the argument list of a DLAMBDA may be a literal atom (as in a conventional LAMBDA) or a list of the form (*NAME TYPE .EXTRAS*). Strictly, this would require a declaration with a SATISFIES clause to take the form (N (NUMBERP (SATISFIES --)) --). However, due to the frequency with which this construction is used, it may be written without the inner set of parentheses, e.g., (N NUMBERP (SATISFIES --) --).

NAME fulfills the standard function of a parameter, i.e., providing a name to which the value of the corresponding argument will be bound.

TYPE is either a Decl package type name or type expression. When the DLAMBDA is entered, its arguments will be evaluated and bound to the corresponding argument names, and then, after all the argument names have been bound, the declarations will be checked. The type checking is delayed so that SATISFIES predicates can include references to other variables bound by the same DLAMBDA. For example, one might wish to define a function whose two arguments are not only both required to be of some given type, but are also required to satisfy some relationship (e.g., that one is less than the other).

EXTRAS allows some additional properties to be attached to a variable. One such property is the accessibility of *NAME* outside the current lexical scope. Accessibility specifications include the atoms LOCAL or SPECIAL, which indicate that this variable is to be compiled so that it is either a LOCALVAR or a SPECVAR, respectively. This is illustrated by the following example:

```
[DLAMBDA ((A LISTP SPECIAL)
          (B FIXP LOCAL))
          ...]
```

A more informative equivalent to the SPECIAL key word is the USEDIN form, the tail of which can be a list of the other functions that are expected to have access to the variable.[1]

```
[DLAMBDA ((A LISTP (USEDIN FOO FIE))
       (B FIXP LOCAL))
       ...]
```

EXTRAS may also include a comment in standard format, so that descriptive information may be given where a variable is bound:

```
[DLAMBDA ((A LISTP (USEDIN FOO FIE)   (* This is an important variable))
          (B FIXP LOCAL))
          ...]
```

As mentioned earlier, the value returned by a DLAMBDA can also be declared, by means of the pseudo-variable RETURNS. The RETURNS declaration is just like other DLAMBDA declarations, except (1) in any SATISFIES predicate, the value of the function is referred to by the distinguished name VALUE; and (2) it makes no sense to declare the return value to be LOCAL or SPECIAL.

DPROG

Just as DLAMBDA resembles LAMBDA, DPROG is analogous to PROG. As for an ordinary PROG, a variable binding may be specified as an atom or a list including an initial value form. However, a DPROG binding also allows *TYPE* and *EXTRAS* information to appear following the initial value form. The format for these augmented variable bindings is (*NAME INITIALVALUE TYPE .EXTRAS*).

The only difference between a DPROG binding and a DLAMBDA binding is that the second position is interpreted as the initial value for the variable. Note that if the user wishes to supply a type declaration for a variable, an initial value *must* be specified. The same rules apply for the interpretation of the type information for DPROGs as for DLAMBDAs, and the same set of optional *EXTRA*s can be used. DPROGs may also declare the type of the value they return, by specifying the pseudo-variable RETURNS.

Just as for a DLAMBDA, type tests in a DPROG are not asserted until after all the variables have been bound, thus permitting predicates to refer to other variables being bound by this DPROG. If NIL appears as the initial value for a binding (i.e., the atom NIL actually appears in the code, not simply an expression that evaluates to NIL) the initial type test will be suppressed, but subsequent type tests, e.g., following a SETQ, will still be performed.

A common construct in Lisp is to bind and initialize a PROG variable to the value of a complicated expression in order to avoid recomputing it, and then to use this value in initializing other PROG variables, e.g.

```
[PROG ((A EXPRESSION))

      (RETURN (PROG ((B... ( A...))

                     (C... ( A... )))

                    ...]
```

The ugliness of such constructions in conventional Lisp often tempts the programmer to loosen the scoping relationships of the variables by binding them all at a single level and using SETQ's in the body of the PROG to establish the initial values for variables that depend on the initial values of other variables, e.g.,

```
[PROG ((A  EXPRESSION) B C)

      (SETQ B (...A... ))

      (SETQ C ( ...A... ))

      ...]
```

In the Decl package environment, this procedure undermines the protection offered by the type mechanism by encouraging the use of uninitialized variables. Therefore, the DPROG offers a syntactic form to encourage more virtuous initialization of its variables. A DPROG variable list may be segmented by occurrences of the special atom THEN, which causes the binding of its variables in stages, so that the bindings made in earlier stages can be used in later ones, e.g.,

```
[DPROG ((A (LENGTH FOO) FIXP LOCAL)

        THEN (B (SQRT A) FLOATP)

        THEN (C (CONS A B) LISTP))

        ...]
```

Each stage is carried out as a conventional set of DPROG bindings (i.e., simultaneously, followed by the appropriate type testing). This layering of the bindings permits one to gradually descend into a inner scope, binding the local names in a very structured and clean fashion, with initial values type-checked as soon as possible.

DECLARATIONS IN ITERATIVE STATEMENTS

The CLISP iterative statement provides a very useful facility for specifying a variety of PROGs that follow certain widely used formats. The Decl package allows declarations to be made for the scope of an iterative statement via the DECLARE CLISP (I.S. operator). DECLARE can appear as an operator anywhere in an iterative statement, followed by a list of declarations, for example:

(for J from 1 to 10 declare (J FIXP) do. . .

Note that DECLARE declarations do not create bindings, but merely provide declarations for existing bindings. For this reason, an initial value cannot be specified and the form of the declaration is the same as that of DLAMBDAs, namely create (*NAME TYPE . EXTRAS*).

Note that variables bound *outside* of the scope of the iterative statement, i.e., a variable used freely in the I.S., can also be declared using this construction. Such a declaration will only be in effect for the scope of the iterative statement.

DECLARING A VARIABLE FOR A RESTRICTED LEXICAL SCOPE

The Decl package also permits declaring the type of a variable over some restricted portion of its existence. For example, suppose the variable X is either a fixed or floating number, and a program branches to treat the two cases separately. On one path X is known to be fixed, whereas on the other it is known to be floating. The Decl package DPROGN construct can be used in such cases to state the type of the variable along each path. DPROGN is exactly like PROGN, except that the second element of the form is interpreted as a list of DLAMBDA format declarations. These declarations are added to any existing declarations in the containing scope, and the composite declaration (created using the ALLOF type expression), is considered to hold throughout the lexical scope created by the DPROGN. Thus, our example becomes:

```
(if (FIXP X)

then (DPROGN ((X FIXP))...else (DPROGN ((X FLOATP)) ...))
```

Like DPROG and DLAMBDA, the value of a DPROGN may also be declared, using the pseudo-variable RETURNS.

DPROGN may be used not only to restrict the declarations of local variables, but also to declare variables that are being used freely. For example, if the variable A is used freely inside a function but is known to be FIXP, this fact could be noted by enclosing the body of the function in (DPROGN ((A FIXP FREE)) *BODY*). Instead of FREE, the more specific construction (BOUNDIN *FUNCTION1 FUNCTION 2*. . .) can be used. This not only states that the variable is used freely but also gives the names of the functions that might have provided this binding.[2]

Since the DPROGN form introduces another level of parenthesization, which results in the enclosed forms being prettyprinted indented, the Decl package also permits such declarations to be attached to their enclosing DLAMBDA or DPROG scopes by placing a DEC expression, e.g., (DECL (A FIXP (BOUNDIN FUM)), before the first executable form in that scope. Like DPROGN's, DECL declarations use DLAMBDA format.

DECLARING THE VALUES OF EXPRESSIONS

The Decl package allows the value of an arbitrary form to be declared with the Decl construct THE. A THE expression is of the form (THE *TYPE . FORMS*), e.g., (THE FIXP (FOO X)). FORMS are evaluated in order, and the value of the *last* one is checked to see if it satisfies *TYPE*, a type name or type expression. If so, its value is returned, otherwise a declaration fault occurs.

ASSERTIONS

The Decl package also allows for checking that an arbitrary predicate holds at a particular point in a program's execution, e.g., a condition that must hold at function entry but not throughout its execution. Such predicates can be checked using an expression of the form (ASSERT *FORM1 FORM2*), in which each *FORM1* is either a list (which will be evaluated) or a variable (whose declaration will be checked). Unless all elements of the ASSERT form are satisfied, a declaration fault will take place.

ASSERTing a variable provides a convenient way of verifying that the value of the variable has not been improperly changed by a lower function. Although a similar effect could be achieved for predicates by explicit checks of the form (OR *PREDICATE* (SHOULDNT)), ASSERT also provides the ability both to check that a variable's declaration is currently satisfied and to remove its checks at compile time without source code modification (see COMPILEIGNOREDECL).

USING TYPE EXPRESSIONS AS PREDICATES

The Decl package extends the Record package TYPE? construct so that it accepts decltypes, as well as record names, e.g., (TYPE? (FIXP (SATISFIES (ILESSP VALUE 0))) EXPR). Thus, a TYPE? expression is exactly the same as a THE expression except that, rather than causing a declaration fault, TYPE? is a predicate that determines whether or not the value satisfies the given type.

ENFORCEMENT

The Decl package is a ''soft'' typing system—that is, the data objects themselves are not inherently typed. Consequently, declarations can only be enforced within the lexical scope in which the declaration takes place, and then only in certain contexts. In general, changes to a variable's value such as those resulting from side effects to embedded structure (e.g., RPLACA, SETN, etc.) or free variable references from outside the scope of the declaration cannot be, and therefore are not, enforced.

Declarations *are* enforced, i.e., checked, in three different situations: when a declared variable is bound to some value or rebound with SETQ or SETQQ, when a declared expression is evaluated, and when an ASSERT expression is evaluated. In a binding context, the type check takes place *after* the binding, including any user-defined behavior specified by the type's binding function. Any failure of the declarations causes a break to occur and an informative message to be printed. In that break, the name to which the declaration is attached (or *VALUE* if no name is available) will be bound to the offending value. Thus, in the FACT T example above, N would be bound to T. The problem can be repaired either by returning an acceptable value from the break via the RETURN command, or by assigning an acceptable value to the offending name and returning from the break via an OK or GO command. The unsatisfied declaration will be reasserted when the computation is continued, so an unacceptable value will be detected.[3]

The automatic enforcement of type declarations is a very flexible and powerful aid to program development. It does, however, exact a considerable run-time cost because of all the checking involved. Factors of two to ten in running speed are not uncommon, especially where low-level, frequently used functions employ type declarations. As a result, it is usually desirable to remove the declaration enforcement code when the system is believed to be bug-free and performance becomes more central. This can be done with the variable COMPILEIGNOREDECL.

COMPILEIGNOREDECL [Variable]

Setting the value of the variable COMPILEIGNOREDECL to T (initially NIL) instructs the compiler not to insert declaration enforcement tests in the compiled code. More selective removal can be achieved by setting COMPILEIGNOREDECL to a list of function names. Any function whose name is found on this list is compiled without declaration enforcement.

IGNOREDECL. *VAL* [File Com]

Declaration enforcement may be suppressed selectively by a file using the IGNOREDECL file package command. If this appears in a file's file commands, it redefines the value of COMPILEIGNOREDECL to VAL for the compilation of this file only.

Note: The period in the IGNOREDECL file package command is significant. To set COMPILEIGNOREDECL to T, use (IGNOREDECL . T), not (IGNOREDECL T).

DECLTYPES

A Decl package type, or decltype, specifies a subset of data values to which values of this type are restricted. For example, a ''positive number'' type might be defined to include only those values that are numbers and greater than zero. A type may also specify how certain operations, such as assignment or binding (see BINDFN), are to be performed on variables declared to be of this type.

The inclusion relations among the sets of values that satisfy the different types define a natural partial ordering on types, bound by the universal type ANY (which all values satisfy) and the empty type NONE (which no value satisfies). Each type has one or more *supertypes* (each type has at least ANY as a supertype) and one or more subtypes (each type has at least NONE as a subtype). This structure is important to the user of Decl as it provides the framework in which new types are defined. Typically, much of the definition of a new type is defaulted, rather than specified explicitly. The definition will be completed by inheriting attributes which are shared by all its immediate supertypes.

An initial set of decltypes that defines the Interlisp built-in data types and a few other commonly used types is provided. Thereafter, new decltypes are created in terms of existing ones using the type expressions described below. For conciseness, such new types can be associated with literal atoms using the function DECLTYPE.

PREDEFINED TYPES

Some commonly used types, such as the Interlisp built-in data types, are already defined when the Decl package is loaded. These types, indented to show subtype-supertype relations, are:[4]

```
ANY

  ATOM             LST

     ARRAYP       STRINGP     FUNCTION    STACKP

    LITATOM          ALIST     HARRAYP

      NIL            LISTP        READTABLEP

    NUMBERP

      FIXP

        LARGEP

        SMALLP

      FLOATP


                                    NONE
```

Note that the definition of LST causes NIL to have multiple supertypes, i.e., LITATOM and LST, reflecting the duality of NIL as an atom and a (degenerate) list.

In addition, declarations made using the Record package also define types that are attached as subtypes to an appropriate existing type (e.g., a TYPERECORD declaration defines a subtype of LISTP, a DATATYPE declaration a subtype of ANY, etc.) and may be used directly in declaration contexts.

TYPE EXPRESSIONS

Type expressions provide convenient ways for defining new types in terms of modifications to, or compositions of one or more existing types.

(MEMQ *VALUE1. . .VALUE N*) [Type Expression]

Specifies a type whose values can be any one of the fixed set of elements VALUE 1. . .VALUE N. For example, the status of a device might be represented by a datum restricted to the values BUSY and

FREE. Such a ‘‘device status’’ type could be defined via (MEMQ BUSY FREE). The new type will be a subtype of the narrowest type that all of the alternatives satisfy (e.g., the "device status" type would be a subtype of LITATOM). The membership test uses EQ if this supertype is a LITATOM; EQUAL otherwise. Thus, lists, floating point numbers, etc., can be included in the set of alternatives.

(ONEOF *TYPE 1. . .TYPE N*) [Type Expression]

Specifies a type that is the union of two or more other types. For example, the notion of a possibly degenerate list is something that is either LISTP or NIL. Such a type can be (and the built-in type LST in fact is) defined simply as (ONEOF NIL LISTP). A union data type becomes a supertype of all of the alternative types specified in the ONEOF expression, and a subtype of their lowest common supertype. The type properties of a union type are taken from its alternative types if they all agree, otherwise from the supertype.

(ALLOF *TYPE 1. . .TYPE N*) [Type Expression]

Specifies a type that is the intersection of two or more other types. For example, a variable may be required to satisfy both FIXP and also some type that is defined as (NUMBERP (SATISFIES *PREDICATE*)). The latter type will admit numbers that are not FIXP, i.e., floating point numbers; the former does not include *PREDICATE*. Both restrictions can be obtained by using the type (ALLOF (NUMBERP (SATISFIES *PREDICATE*)) FIXP).[5]

(OF *AGGREGATE* OF *ELEMENT*) [Type Expression]

Specifies *DECLaggregate,* a type that is an aggregate of values of some other type (e.g., list of numbers, array of strings, etc.). *AGGREGATE* must be a type that provides an EVERYFN property. The EVERYFN is used to apply an arbitrary function to each of the elements of a datum of the aggregate type, and check whether the result is non-NIL for each element. *ELEMENT* may be any type expression. For example, the type ‘‘list of either strings or atoms’’ can be defined as (LISTP OF (ONEOF STRINGP ATOM)). The type test for the new type will consist of applying the type test for *ELEMENT* to each element of the aggregate type using the EVERYFN property. The new type will be a subtype of its aggregate type.[6]

(SATISFIES *TYPE* (SATISFIES *FORM 1. . .FORM N*)) [Type Expression]

Specifies a type whose values are a subset of the values of an existing type. The type test for the new type will first check that the base type is satisfied, i.e., that the object is a member of TYPE, and then evaluate *FORM 1. . .FORM N*. If each form returns a non-NIL value, the type is satisfied.

The value that is being tested may be referred to in *FORM 1. . . FORM N* by either (a) the variable name if the type expression appears in a binding context such as DLAMBDA or DPROG, (b) the distinguished atom ELT for a SATISFIES clause on the elements of an aggregate type, or (c) the distinguished atom VALUE, when the type expression is used in a context where no name is available (e.g., a RETURNS declaration). For example, one might declare the program variable A to be a negative integer via (FIXP (SATISFIES (MINUSP A))) or declare the value of a DLAMBDA to be of type ((ONEOF FIXP FLOATP) (SATISFIES (GREATERP VALUE 25))).

Note that more than one SATISFIES clause may appear in a single type expression attached to different alternatives in a ONEOF type expression, or attached to both the elements and the overall structure of an aggregate. For example,

```
[LISTP OF [FIXP (SATISFIES (ILEQ ELT (CAR VALUE]

          (SATISFIES (ILESSP (LENGTH VALUE) 7]
```

specifies a list of less than seven integers each of which is no greater than the first element of the list.

(SHARED *TYPE*) [Type Expression]

Specifies DECLshared, a subtype of *TYPE,* with default binding behavior, i.e., the binding function (see BINDFN), if any, will be suppressed.[7] For example, if the type FLOATP were redefined so that DLAMBDA and DPROG bindings of variables that were declared to be FLOATP copied their initial values (e.g., to allow SETNs to be free of side effects), then variables declared (SHARED FLOATP) would be initialized in the normal fashion, without copying their initial values.

NAMED TYPES

Although type expressions can be used in any declaration context, it is often desirable to save the definition of a new type if it is to be used frequently, or if a more complex specification of its behavior is to be given than is convenient in an expression. The ability to define a named type is provided by the function DECLTYPE.

(DECLTYPE *TYPENAME TYPE PROP1 VAL1*
PROPN VALN) [Function]

NLambda, nospread function. *TYPENAME* is a literal atom, *TYPE* is either the name of an existing type or a type expression, and *PROP 1, VAL 1...PROP N,VAL N* is a specification (in property list format) of other attributes of the type. DECLTYPE derives a type from *TYPE*, associates it with *TYPENAME*, and then defines any properties specified with the values given.

The following properties are interpreted by the Decl package.[8] Each of these properties can have as its value either a function name or a LAMBDA expression.

TESTFN [Property]

will be used by the Decl package to test whether a given value satisfies this type. The type is considered satisfied if *FN* applied to the item is non-NIL. For example, one might define the type INTEGER with TESTFN FIXP.[9]

EVERYFN [Property]

EVERYFN specifies a mapping function that can apply a functional argument to each ‘‘element’’ of an instance of this type, and which will return NIL unless the result of every such application was non-NIL. *FN* must be a function of two arguments: the aggregate and the function to be applied. For example, the EVERYFN for the built-in type LISTP is EVERY. The Decl package uses the EVERYFN property of the aggregate type to construct a type test for aggregate type expressions. In fact, it is the presence of an EVERYFN property that allows a type to be used as an aggregate type.[10]

BINDFN [Property]

BINDFN is used to compute from the initial value supplied for a DLAMBDA or DPROG variable of this type, the value to which the variable will actually be initialized. *FN* must be a function of one argument that will be applied to the initial value, and which should produce another value which is to be used to make the binding.[11] For example, a BINDFN could be used to bind variables of some type so that new bindings are copies of the initial value. Thus, if FLOATP were given the BINDFN FPLUS, any variable declared FLOATP would be initialized with a new floating box, rather than sharing with that of the original initial value.[12]

SETFN [Property]

is used for performing a SETQ or SETQQ of variables of this type. *FN* is a function of two arguments, the name of the variable and its new value. A SETFN is typically used to avoid the allocation of storage for intermediate results. Note that the SETFN is not the mechanism for the enforcement of type compatibility, which is checked *after* the assignment has taken place. Also note that not all functions that can change values are affected: in particular, SET and SETN are not.

MANIPULATING NAMED TYPES

DECLTYPES is a file package type. Thus all of the operations relating to file package types, e.g., GETDEF, PUTDEF, EDITDEF, DELDEF, SHOWDEF, etc., can be performed on decltypes.[13]

The file package command, DECLTYPES , is provided to dump named decltypes symbolically. They will be written as a series of DECLTYPE forms that will specify only those fields that differ from the corresponding field of their supertype(s). If the type depends on any unnamed types, those types will be dumped (as a compound type expression), continuing up the supertype chain until a named type is found. Care should be exercised to ensure that enough of the named type context is dumped to allow the type definition to remain meaningful.

The functions GETDECLTYPEPROP and SETDECLTYPEPROP, defined analogously to the property list functions for atoms, allow the manipulation of the properties of named types. Setting a property to NIL with SETDECLTYPEPROP removes it from the type.

RELATIONS BETWEEN TYPES

The notion of equivalence of two types is not well defined. However, type equivalence is rarely of interest. What is of interest is type *inclusion*, i.e., whether one type is a supertype or subtype of another. The predicate COVERS can be used to determine whether the values of one type include those of another.

(COVERS *HI LO*) [Function]

COVERS is T if *HI* can be found on some (possibly empty) supertype chain of *LO*; else NIL. Thus, (COVERS 'FIXP (DECLOF 4))= T, even though the DECLTYPE of four is SMALLP, not FIXP. The extremal cases are the obvious identities:

```
(COVERS 'ANY ANYTYPE) = (COVERS ANYTYPE 'NONE) = (COVERS X ) for any type  X
= T.
```

COVERS allows declaration-based transformations of a form that depend on elements of the form being of a certain type to express their applicability conditions in terms of the weakest type to which they apply, without explicit concern for other types that may be subtypes of it. For example, if a particular transformation is to be applied whenever an element is of type NUMBERP, the program that applies that transformation does not have to check whether the element is of type SMALLP, LARGEP, FIXP, FLOATP, etc., but can simply ask whether NUMBERP COVERS the type of that element.

The elementary relations among the types, out of which arbitrary traversals of the type space can be constructed, are made available via:

(SUBTYPE *TYPE)* [Function]

Returns the list of types that are *immediate* subtypes of *TYPE*.

(SUPERTYPES *TYPE*) [Function]

Returns the list of types that are *immediate* supertypes of *TYPE*.

THE DECLARATION DATA BASE

One of the primary uses of type declarations is to provide information that other systems can use to interpret or optimize code. For example, one might choose to write all arithmetic operations in terms of general functions like PLUS and TIMES and then use variable declarations to substitute more efficient, special-purpose code at compile time based on the types of the operands. To this end, a data base of declarations is made available by the Decl package to support these operations.

(DECLOF *FORM*) [Function]

Returns the type of *FORM* in the current declaration context. If *FORM* is an atom, DECLOF will look up that atom directly in its data base of current declarations. Otherwise, DECLOF will look on the property list of (CAR *FORM*) for a DECLOF property, as described below. If there is no DECLOF property, DECLOF will check if (CAR *FORM*) is one of a large set of functions of known result type (e.g., the arithmetic functions). Failing that, if (CAR *FORM*) has a MACRO property, DECLOF will apply itself to the result of expanding (with EXPANDMACRO), the macro definition. Finally, if *FORM* is a Lisp program element that DECLOF ‘‘understands’’ (e.g., a COND, PROG, SELECTQ, etc.), DECLOF applies itself recursively to the part(s) of the contained form which will be returned as value.[14]

DECLOF [Property]

Allows the specification of the type of the values returned by a particular function. The value of the DECLOF property can be either a type, i.e., a type name or a type expression, or a list of the form (FUNCTION *FN*), where *FN* is a function object. *FN* will be applied (by DECLOF) to the form whose CAR has this DECLOF property on its property list. The value of this function application will then be considered to be the type of the form.

As an example of how declarations can be used to automatically generate more efficient code, consider an arithmetic package. Declarations of numeric variables could be used to guide code generation to avoid the inefficiencies of Interlisp’s handling of arithmetic values. Not only could the generic arithmetic functions be automatically specialized, as suggested above, but by redefining the BINDFN and the SETFN properties for the types FLOATP and LARGEP to reuse storage in the appropriate contexts (i.e., when the new value can be determined to be of the appropriate type), tremendous economies could be realized by not allocating storage to intermediate results that must later be reclaimed by the garbage collector. The Decl package has been used as the basis for several such code optimizing systems.

DECLARATIONS AND MASTERSCOPE

The Decl package notifies MASTERSCOPE about type declarations and defines a new MASTERSCOPE relation, TYPE, which depends on declarations. Thus, the user can ask questions such as ‘‘WHO USES MUMBLE AS A TYPE?,’’ ‘‘ DOES FOO USE FIXP AS A TYPE?,’’ and so on.

END NOTES

1. USEDIN is mainly for documentation purposes, since there is no way for such a restriction to be enforced.

2. Like USEDIN declarations, FREE and BOUNDIN declarations cannot be checked, and are for documentation purposes only.

3. With this exception, assignments to variables from within the break are not considered to be in the scope of the declarations that were in effect when the break took place and so are not checked.

4. LST is defined as either LISTP or NIL. i.e., a list or NIL. The name LST is used because the name LIST is treated specially by CLISP. A LIST is defined as either NIL or a list of elements each of which is of type LISTP.

5. When a value is tested, the component type tests are applied from left to right.

6. The built-in aggregate types are ARRAP, LISTP, LST, and STRINGP (and their subtypes).

7. As no predefined type has a binding function, this is of no concern until the user defines or redefines a type to have a binding function.

8. Actually, any property can be attached to a type, and will be available for use by user functions via the function GETDECLTYPEPROP.

9. Typically, the TESTFN for a type is derived from its type expression, rather than specified explicitly. The ability to specify the TESTFN is provided for those cases where a predicate is available that is much more efficient than that which would be derived from the type expression. For example, the type

SMALLP is defined to have the function SMALLP as its TESTFN, rather than (LAMBDA(DATUM) (AND(NUMBERP DATUM)(FIXP DATUM) (SMALLP DATUM))) as would be derived from the subtype structure.

10. Note that a type's EVERYFN is not used in type tests for that type, but only in type tests for types defined by OF expressions that used this type as the aggregate type. For example, EVERY is not used in defining whether some value satisfies the type LISTP. The Decl package never applies the EVERYFN of a type to a value without first verifying that the value satisfies that type.

11. For a PPROG binding, FN will be applied to no arguments if the initial value is lexically NIL.

12. The BINDFN, if any, associated with a type may be suppressed in a declaration context by creating a subtype with the type-expressing operator SHARED.

13. Deleting a named type could possibly invalidate other type definitions that have the named type as a subtype or supertype. Consequently, the deleted type is simply unnamed and left in the type space as long as it is needed.

14. ''The current declaration context'' is defined by the environment at the time that DECLOF is called. Code-reading systems, such as the compiler and the interpreter, keep track of the lexical scope within which they are currently operating, in particular, which declarations are in effect. Note that (currently) DECLOF does *not* have access to any global data base of declarations. For example, DECLOF does not have information available about the types of arguments of, or the value returned by, a particular function, unless it is currently ''inside'' that function. However, the DECLOF property can be used to inform DECLOF of the type of the value returned by a particular function.

DEFAULTSUBITEMFN

By: Nick Briggs (Briggs.pa@Xerox.com)

The DEFAULTSUBITEMFN module redefines the DEFAULTSUBITEMFN to permit an extended specification of menu subitems. If the CAR of the 4th element of a menu item is the keyword EVAL, the CADR of that 4th element is evaluated and the results used as the subitem specifications. During the evaluation the variables MENU and ITEM are bound respectively to the menu and item of which the EVAL subitem spec is a part. This module is only a stopgap measure until it is possible to easily redefine the BackgroundMenu subitem function, but it will provide this facility on all menus that do not explicity specify a subitem function.

example menu item entries:

```
(foo foo.selected "No help for you!" (EVAL dynamic.foo.subitems))
```

using a variable containing subitems, or

```
(foo foo.selected "No help for you!" (EVAL (compute.foo.subitems))
```

using a function to recompute the subitems.

It is prudent to make the expressions used in the EVAL subitems quite efficient, since they will be called many times.

DIALPHONE

By: Michel Denber (Denber.WBST@Xerox.COM)

INTRODUCTION

DIALPHONE is a simple computer-controlled telephone dialer. It requires a modem connected to the RS-232 port. It should work with a wide variety of modems since it makes no assumption about any return codes from the modem. It lets you dial your phone by typing (or selecting) a number fromthe computer's keyboard. A history list of recently dialed numbers is maintained. It knows how to deal with differences between dialing local extensions, outside calls, and long distance. It also translates phone numbers containing letters back into numbers.

OPERATION

Load DIALPHONE.LCOM from your local LispUsers directory and call (DIALPHONE). The program will ask you for your own phone number. This is used within Xerox for billing long distance calls. If you just enter a CR, the program will not append any number at the "quick-quick-quick-slow" tone when dialing long distance. Next, it will try to attach a menu with a telephone icon to an AddressBook window (if you have the LispUsers package AddressBook loaded), since it makes a useful complement to AddressBook. If you do not have AddressBook loaded, it will prompt you to click in some window where you would like the Dial icon attached. If you click in an area of background, a stand-alone menu will be created. Before dialing a number, check the various paramters described below.

To dial a number, click on the phone icon with the left mouse button. You will see the prompt "Number please:" in the prompt window. You may now either type in a number or shift-select it out of another window. To abort the operation, type a CR without a number. If you select the icon with the middle button, you will get a menu of the last 10 numbers dialed. Listen to the number being dialed over the modem's speaker, then pick up the receiver. The program will make the modem hang up automatically shortly after the number is dialed.

The program converts any letters in the number you give it into numbers. The program is designed for use on a PBX-style system (although it is easy to modify if desired). Numbers less than six digits long are assumed to be internal extensions and are dialed exactly as given. Numbers or 7 or 8 digits are assumed to be local outside calls; the program prefixes a "9" to them. Numbers longer than 8 digits are assumed to be long distance; the program appends your extension to them for billing. Xerox Intelent numbers should be typed in their usual form, starting with "8*". Area codes and exchanges should be separated with dashes, e.g. "716-555-1212".

You can also call the program directly, e.g. (DIALPHONE "555-1212"). It will return the number dialed. If you try to dial a number while the modem port is already in use, the program will print an error message in the Prompt Window and return without dialing.

DIALHISTSIZE [Variable]

This controls how many numbers will appear in the menu you get when you select the phone icon with the middle button. Default = 10.

DIALPREFIX [Variable]

The modem command your modem needs to initiate dialing. Default = "ATDT". Change this to "ATDP" if you do not have touch-tone service.

DIALSUFFIX [Variable]

The modem command your modem needs to terminate a command. Default = <CR> (ASCII 13).

LASTNUMBERDIALED [Variable]

The last number you dialed. initially NIL.

PHONEBILLNUMBER [Variable]

The number the program should use to append to long-distance numbers. Initially "" (null string).

DICTTOOL

By: Maxwell (Maxwell.pa@Xerox.com)

Uses DICTCLIENT, ANALYZER

INTERNAL

INTRODUCTION

DICTTOOL is the user's interface to the Dictionary Server. The Dictionary Server is a prototype of a shared network resource for providing a suite of dictionary-based capabilities to programs running on client workstations. It has on it the American Heritage dictionary, the Word Finder synonym package by Microlytics, a Proofreader, and the WordNerd, a package for searching for words based on their meaning.

(Note: The American Heritage dictionary has been licensed to us by Houghton-Miflin for research purposes only, and so we have not made the Dictionary Server generally available. The Dictionary Server should only be used by people within PARC.)

HOW TO USE DICTTOOL

When you load the DICTTOOL, it automatically adds a new menu item named "Dictionary" to the TEdit menu and the Background menu. The "Dictionary" menu item has three sub-items: "Get Definition", "Get Synonyms", and "Search For Word". Here is how each one works:

Get Definition

If you make a selection in a TEdit document, and then invoke the "Get Definition" command in that document, then DictTool will ask for a confirmation and then fetch the definition for that word from the Dictionary Server, printing it in a separate window. If there is more than one entry in the American Heritage Dictionary for that word, then it will print the definitions one after another. The Dictionary Server knows how to find the root forms of words, and so "breathing" "breathes" and "breathe" will all give you the same entry.

If there is no selection in the TEdit document, or if you deny the confirmation, or if instead of using the TEdit menu you use the Background menu, then DictTool will first prompt you for a word to look up and then fetch its definition. (Since it is very hard to make a null selection with TEdit, DictTool treats a one character selection as meaning "no selection". If you really want to look up a single letter in the dictionary, you can type it in when prompted for a word.)

If you want to look up several definitions at once, separate the entries with semi-colons followed by spaces. (i.e. "camera; photography; motion picture"). Semi-colons are used as delimiters because some of the entries in the American Heritage Dictionary have spaces and commas in them (as in "motion picture"). It also makes it easier to look up words in the output of the WordNerd (see "Search For Word" below).

Get Synonyms

The interface for getting synonyms is exactly the same as the interface for getting definitions. If you make a selection, then DictTool will first confirm the selection and then print out the synonyms in the same window that the definitions are printed in. If you don't make a selection, then DictTool will first prompt you for a word. The format of the information printed out is a series of synonym classes separated by carriage returns. Each synonym class begins with the part of speech that its elements belong to. The elements themselves are separated by commas.

Search For Word

The interface to the WordNerd is a little different from the other interfaces. Instead of typing just one word in, you want to type a list of keywords separated by spaces. For instance, if you were looking for the word for a mechanical model of the solar system, you might type:

> *Type keywords to search on*: mechanical model solar system

The WordNerd then searches for words that have at least two of these keywords in their definitions. The results would be sorted according to the number of keywords found, with the words having the most keywords printed first:

> **mechanical model solar system:** orrery
>
> **mechanical solar system:** mechanism
>
> **model solar system:** planetarium
>
> **mechanical system:** automation; bar1; component; degree of freedom; energy level; hookup; ignition; instrument; key1; linkwork; load; machine; neutral; perpetual motion; quantize; resonance; schematic; servomechanism; shafting; stress; suspension; unit
>
> **solar system:** Copernican; cosmic; Earth; Ganymede3; Jupiter2; Mars2; mercury; Milky Way; nebular hypothesis; Neptune3; Pallas; planet; planetesimal hypothesis; Pluto2; Saturn2; solar battery; space; sun; Uranus2; Venus2; Vesta2

(The numbers after some of the words mean that this is the nth entry of this word in the American Heritage Dictionary.) If there is a word in the list that you want to see the definition for, you can merely select it and get its definition with "Get Definition". In this case you would probably want to know what "orrery" means:

or|re|ry *n., pl.* **-ries.** A mechanical model of the solar system. [After Charles Boyle (1676–1731), fourth Earl of *Orrery*, for whom one was made.]

There is also a mechanism for indicating that two words are synonyms of one another, and hence should not be counted as separate keywords for the purpose of deciding whether a word has the minimum two keywords. All you need to do is put parentheses around the words in question. For instance if you were looking for the word for the little plastic thing on the end of a shoe lace, you might try:

Type keywords to search on: (shoe lace shoelace) (end tip)

And get in return:

shoe+ end+: aglet; fall; heel1; lift; point; quarter; spike1; toe

(A plus at the end of a word indicates a synonym class.)

If you only give the DictTool one word, then it will print out all of the words in the dictionary that have that word in its definition.

Max Words

There are two sub-items in the "Search For Word" sub-menu: "Max Words" and "Min Keywords". The first sub-item, "Max Words", allows the user to specify the maximum number of words that should be returned on each search. DictTool is set up to only return 100 words at a time. If WordNerd finds more than a hundred words, then it truncates the list and indicates how many words it eliminated. If you want to see the words that were eliminated, just make the same request with the same keywords in the same order and the WordNerd will return the next 100 words. (If there is no selection in the document, then DictTool will prompt you with the last set of keywords so that this is easier.) However, if 100 words is too many or too few, you can change it with this menu item or by setting the global variable DictTool.MaxWords.

Min Keywords

DictTool is set up to only return a word if it has at least two of the user’s keywords in its definition. If the user wants, he can raise or lower the minimum as he sees fit. The minimum only comes into play whenever the user gives more keywords than the minimum, otherwise the WordNerd looks for words that have at least one of the keywords in their definition. A minimum of 1 means that only one word has to match. A negative minimum means that the WordNerd will set the minimum relative to the number of keywords given. For instance, a minimum of -1 says that all but one of the keywords have to match for the word to be returned. A minimum of zero means that *all* of the words have to match.

Search For Phrase

The Search For Phrase command returns all of the entries in the American Heritage Dictionary that have a particular phrase in them. It does this with the help of the Search For Word command, which is why it is a sub-command of that command. Whenever you search for a phrase, the dictionary server first uses the Search For Word command to get the list of words in the dictionary that have all of the words of the phrase in it. It then looks up the definition of each of these words, and returns the words that have the phrase in their definition. This can be a very time-consuming operation, so you should use this command sparingly. But if you are concerned about locality and word order, then this command can save you a lot of time.

PROOFREADING

The Dictionary Server also provides proofreading facilities similar to the PROOFREADER package. The interface is exactly the same: there is a "Proofread" menu item on the TEdit menu which produces a special fixed menu for proofreading. The only difference is that all of the proofreading is done remotely on the server. You should only use the Dictionary Server for proofreading small documents; if you are going to do a lot of proofreading, it is better to use the PROOFREADER. (For more documentation on how to proofread, see PROOFREADER.)

DIGI-CLOCK

By: Keith Mountford (Mountford.AISNorth@Xerox.Com)

INTRODUCTION

DIGI-CLOCK is a digital clock which allows you to keep track of the time in multiple time zones.

STARTING DIGI-CLOCK

Loading DIGI-CLOCK will kill any existing DIGI-CLOCK process and restart the clock. Once the clock is loaded it can be restarted by typing (DIGI-CLOCK) or (DIGI-CLOCK T). The second of these restarts the clock from scratch, rebuilding everything; the first, simply restarts the process and does not undo any changes made to the clock. Left buttoning in the window causes the clock to update itself. The clock updates itself approximately once a minute.

CHANGING DIGI-CLOCK

The clock font, the time, the local time zone, the alarm, the alarm mode (loud or quite), the clock mode (12 or 24 hour) are all settable from the middle button menu. This menu also allows you to add clocks for other time zones. The auxilliary clocks also have middle button menus which allow you to set the time zone for that window and edit the time zone heading. The default is for all of the auxilliary clocks have the same font and changing the font in one changes the font in all of them unless the submenu item "Set Aux Clock Font In Just This Window" is slected. Selecting "Shape to Fit" will reshape the clock windows to their minimum size.

If the menu font options are not sufficient you can set the global variables *DC-FONT* and *DC-AUXW-FONT*. The date format is bound to the variable *DC-DATEFORMAT* and can be changed by editing or setting this variable. The clock does not deal with seconds gracefully in 12-hour mode and it will not allow NUMBER.OF.MONTH in 12-hour mode. The regional time zones are stored on the global list *DC-TIME-ZONE-LIST*.

SETTING DIGI-CLOCK

Choosing "Set Time" from the middle button menu, brings up a menu which allows you to set the time.

SETTING THE DIGI-CLOCK ALARM

DIGI-CLOCK includes an alarm clock which can be set to any number of dates in any order. The alarm stores a brief message to remind you why the alarm was set. To set the alarm choose the "Set Alarm" middle button option. Once you have set the time, the clock will prompt you for a message. This message can be longer than the window, but only one line long. When the alarm rings, the window will shape to fit the message.

The alarm calls the function RINGBELLS once a minute until the alarm is turned off, which can be annoying. To run the alarm in quiet mode, select Quiet Alarm from the middle button menu. Selecting Quiet Alarm changes this menu option to Loud Alarm and sets the alarm to run in quiet mode. Selecting Loud Alarm will toggle the alarm back to its original noisy setting.

To unset the alarm, select "Delete Alarm Setting" from the middle-button menu and then select the time you want deleted from the pop-up menu.

To turn the alarm off, select "Turn Alarm Off" from the middle-button menu.

DINFO

By: Doug Cutting (Cutting.PA@Xerox.COM)

Uses: GRAPHER, TEDIT

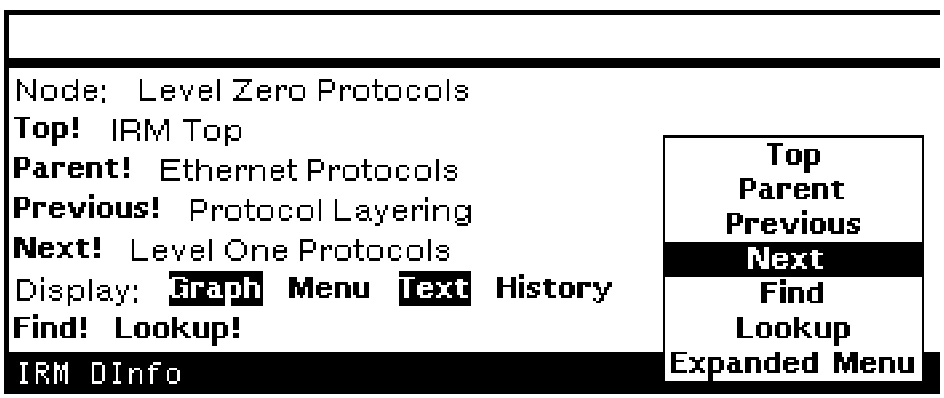
This document last edited on October 7, 1987.

INTRODUCTION

DInfo is a system for browsing graph structured documentation. Graphs for the *Interlisp-D Reference Manual* and the *Xerox Quintus Prolog Manual* are available, as are tools for creating and editing new graphs.

USER INTERFACE

Selecting DInfo from the background menu will pop up a menu listing the available graphs (see DINFO.GRAPHS below). Selecting one of these items will bring up a browser on that graph. Most interaction with DInfo is done through menus at the top of each graph browser which look like this:



The Expanded Menu may be closed independently of the graph browser, and re-opened by selecting **Expanded Menu** from the title bar menu. Except for this command, the commands in the title bar menu are identical to their counterparts in the Expanded Menu, so only the latter commands are documented.

Next to **Node:**, **Top!**, **Parent!**, **Previous!**, and **Next!** are printed the names of the node currently being visited, the name of the top node in the graph, the name of the node previous to the current node, and the name of the node next to the previous node, respectively. **Top!**, **Parent!**, **Previous!**, and **Next!** are also commands as follows:

Top! [DInfo Menu Command]

Visits the top node in the graph.

Parent! [DInfo Menu Command]

Visits the parent of the current node.

Previous! [DInfo Menu Command]

Visits the node previous to the current node.

Next! [DInfo Menu Command]

Visits the node following the current node.

Previous! and **Next!** thus provide sequential access to the graph.

The Display: toggles control what will be displayed when a node is visited:

Graph [DInfo Menu Toggle]

toggles display of a Grapher display of the graph local to the current node. Selecting a node in this display will visit the corresponding node in the graph.

Menu [DInfo Menu Toggle]

Toggles display of a menu of subnodes of the current node. If the current node has no subnodes no menu will be displayed. Selecting an item in the subnode Menu will visit that node in the graph.

Text [DInfo Menu Toggle]

Toggles display of the text of the current node. Turning this off will speed up the visiting of nodes considerably, useful when searching for a particular node.

History [DInfo Menu Toggle]

Toggles display of a menu containing the history of nodes visited. Selecting an item from this menu will revisit that node.

Lookup! [DInfo Menu Command]

If selected with the left mouse button prompts for a term and then calls the LOOKUPFN of the graph with it. Using the middle button will re-call the LOOKUPFN with whatever was used previously. This is intended for the lookup of terms in a graph-dependent index. In the case of the *Interlisp-D Reference Manual* DInfo Graph, Lookup! will lookup a term in the index of the IRM, and then visit the node containing the reference to this term.

(DINFO *GRAPH.OR.FILE WINDOW.OR.REGION* — —) [Function]

Starts a DInfo browser on GRAPH.OR.FILE in WINDOW.OR.REGION. If GRAPH.OR.FILE is NIL, an empty graph will be created. If WINDOW.OR.REGION is NIL, the user will be prompted for a window or region.

GLOBAL VARIABLES

DINFOMODES [Variable]

Determines which of the toggles will be selected when DInfo is initially started; it should be a list with recognized members being GRAPH, MENU, TEXT, and HISTORY. Default is (GRAPH TEXT).

DINFO.HISTORY.LENGTH [Variable]

Determines the maximum length of DInfo’s history. Default is 20.

DINFO.GRAPHS [Variable]

Determines the contents of the menu raised by selecting DInfo from the background menu. Should be a menu-items style list where when the CADR of an item is evaluated it returns either a Dinfo Graph (a DINFOGRAPH record) or the name of a file containing a DInfo Graph (i.e, something suitable for passing to DInfo as the GRAPH.OR.FILE arguement).

INTERNALS

The following information is included for the programmer interested in adding alternate graphs to DInfo.

DINFONODE [Record]

Contains the following fields:

ID Unique identifier for node in graph, ala GRAPHNODE field NODEID. Note that EQ is used for checking identity of nodes.

LABEL The print name of a node. Analagous to the GRAPHNODE field NODELABEL.

FILE The file containing the documentation for this node. Should not generally include HOST and DIRECTORY fields as DInfo will default these (assuming all documentation files are on one directory, see below).

FROMBYTE Byte number in FILE where the documentation for this node begins.

TOBYTE Byte number in FILE where the documentation for this node ends.

DInfo uses OPENTEXTSTREAM to display its files, and thus any TEdit file can be included. Note that if a file has any formatting (image objects in particular), the byte number of a character in a file is not necessarily the same as the TEdit character number of that character.

PARENT The ID field of the node parent to this node.

CHILDREN A list of the ID's of the subnodes of this node.

NEXTNODE The ID of the next node in the graph.

PREVIOUSNODE The ID of the node previous to this node in the graph.

USERDATA Unused. Note that there is no special access function for this field as, for example, WINDOWPROP is for the USERDATA field of a WINDOW. This field is left open for use by implementors for whatever they see fit.

DINFOGRAPH [Data Type]

Contains the following fields of interest to the implementor:

NAME The name of the graph. Note that when DInfo reads a graph from a file (with DINFO.READ.GRAPH) this field is set to the NAME field of the file name the graph is read from.

NODELST The list of nodes in the graph. Each node should be a DINFONODE record.

TOPNODEID The ID field of the root, or top node of the graph.

WINDOW The main window of the graph browser.

CURRENTNODE Used by DInfo to keep track of where in the graph it is.

DEFAULTHOST Used if no host is specified in the FILE field of a node.

DEFAULTDEVICE Used if no device is specified in the FILE field of a node.

DEFAULTDIR Used if no directory is specified in the FILE field of a node.

Note that DEFAULTHOST, DEFAULTDEVICE and DEFAULTDIR are set when a DINFOGRAPH is read from a file (by DINFO.READ.GRAPH) to the host, device and directory of that file.

TEXTPROPS Will be passed as the PROPS argument to OPENTEXTSTREAM when the file for a node in the graph is displayed. This feature can be to used to fake some formatting .

LOOKUPFN Will be called when the user selects Lookup! from DInfo's Expanded Menu with two arguments: The string to look up, and the current DInfo graph.

MENUFN Called when the middle mouse is depressed in the title bar of a graph's window . If not specified, DINFO.DEFAULT.MENU will be used. Passed one argument of the current DInfo graph. DINFO.EDIT.MENU is a MENUFN that allows editing of DInfo graphs. Selecting >>Empty Graph<< from the menu raised by selecting DInfo from the Background Menu will start DInfo on a an empty graph with this MENUFN.

FREEMENUITEMS DInfo's Expanded Menu is implemented as a FreeMenu. This property holds a list of FreeMenu item descriptions suitable for passing to FREEMENU (see FreeMenu documentation). This list will be appended onto the bottom row of buttons (**Find!** and **Lookup!)** whenever a FreeMenu is created for this graph.

USERDATA Accessed by the macro DINFOGRAPHPROP. See below.

(DINFOGRAPHPROP *GRAPH PROP VALUE*) [Macro]

If VALUE is not specified, will return the PROP of GRAPH. PROP can be either a real DINFOGRAPH field or something in the USERDATA field. If VALUE is specified it will be put in GRAPH at PROP. Note that in this case it will return the *new value* stored in, not the previous value, as many other Interlisp-D access functions do.

(DINFO.UPDATE *GRAPH NODE SEL*) [Function]

Will visit NODE in GRAPH. *NODE,* if specified*,* should be a DINFONODE record which is in the NODELST of GRAPH. *SEL* is used by DInfo's *Interlisp-D Reference Manual* lookup facility, and should be useful in implementing other lookup facilities. *SEL* determines what in the TEXT of this node will be selected. *SEL* should be a string or a list of the format (*NAME X*) where *NAME* is the name of the selection, *X* is the character number in the text of *NODE* to be scrolled to. If SEL is a string, the string will be searched for in the text of NODE and selected. This is useful for the lookup of terms.

(DINFOGRAPH *WINDOW*) [Function]

Return the DINFOGRAPH associated with window. Note that the pointer from the window to the graph is destroyed when the window is closed to remove circularity. For this reason it is better to keep a handle on the DINFOGRAPH and use (DINFOGRAPHPROP <DInfoGraph> 'WINDOW) when you need the WINDOW.

(DINFO.READ.GRAPH *FILE QUIETFLG*) [Function]

Reads a file written by DINFO.WRITE.GRAPH, and returns the DINFOGRAPH contained therein. If QUIETFLG is non-NIL, nothing will be printed out while reading.

(DINFO.WRITE.GRAPH *GRAPH FILE*) [Function]

Writes GRAPH to FILE such that it can be read by DINFO.READ.GRAPH.

(DINFO.READ.KOTO.GRAPH *GRAPH FILE*] [Function]

Reads a file written by Koto DINFO.WRITE.GRAPH and returns a Lyric DINFOGRAPH. Thus:

```
(DINFO.WRITE.GRAPH
  (DINFO.READ.KOTO.GRAPH <file1> T)
  <file2>)
```

will convert the Koto format DINFOGRAPH in `<file1>` to a Lyric format DINFOGRAPH in `<file2>`.

DIR-TREE

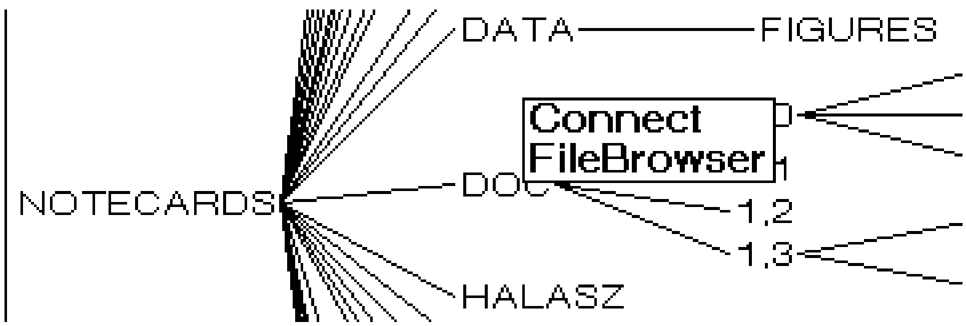
Dir-Tree builds directory trees using the grapher package.

It works on IFS, XNS, and NFS devices from Medley.

Loading Dir-Tree puts a menu option on your background menu.

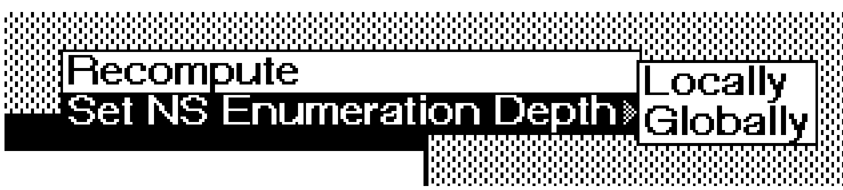
Dir-Tree can take awhile to compute a large directory. Do not give it a /users directory to graph.

From the directory window you can left button on a node and bring up a menu to Connect to the directory or call FileBrowser on the directory.



At the moment shift-select from the graph does not work but you can achieve the same effect by clicking the middle mouse button on a node.

Holding the left mouse button down in the title bar brings up the recompute option. On NS servers it also allows you to set the Enumeration Depth.



I make no pretenses that Dir-Tree is anything like a finished product. However, it has been satisfactorily working for me in its present state for over a month now (Jan '89).

I have been asked when it will be available. Since I will not have any time to work on it until the NoteCards doc is done, those who are interested are welcome to use it in its present state.

KNOWN PROBLEMS:

When using it from a Sun DO NOT use the {UNIX} device. This hopelessly confuses Dir-Tree.

When specifying the root directory on a Sun, you must get the case right. Dir-Tree will still generate a graph but it will only go down one level, and it will not correctly truncate the root directory name in the graph window.

Keith Mountford.

DOC-OBJECTS

Johannes A. G. M. Koomen
(Koomen.wbst@Xerox.com or Koomen@CS.Rochester.edu)

Uses: TEDIT, IMAGEOBJ, DATEFORMAT-EDITOR

This document last edited on October 27, 1987.

DESCRIPTION

DOC-OBJECTS is a generic, extensible interface for including image objects in TEdit documents. It hooks into TEdit by an extra entry on TEdit's middle button menu, as well as by redefining what happens on typing CTRL-O. Clicking the menu entry or typing CTRL-O brings up an *Objects* menu. Selecting an object causes an instance of the designated object to be inserted in the document at the position of the caret. Clicking outside the Objects menu has no effect. DOC-OBJECTS comes with a set of predefined Document Objects, which are described below. Additional Objects can easily be added to the Objects menu.

Predefined Objects

Time Stamp [Document Object]

A TimeStamp reflects the date the document containing it was last PUT into a file. Each PUT causes a TimeStamp to be updated. Clicking the mIddle button over a TimeStamp brings up a DateFormat editor. The TimeStamp can be given any appearance consistent with the function DATEFORMAT (see IRM, Section 12.5). The object following the next colon is a TimeStamp object for this file: 15 Sep 1988 18:11 PDT (Thursday). Individual characters of a TimeStamp cannot be altered by TEdit, but a TimeStamp can be given arbitrary TEdit Looks. The DATEFORMAT-EDITOR package is automatically loaded by the DOC-OBJECTS package.

File Stamp [Document Object]

A FileStamp reflects the name of the file into which the document containing it was last PUT. Each PUT causes a FileStamp to be updated. It cannot be edited. A FileStamp is initially displayed as "`-- not yet filed --`".

Include [Document Object]

This document object is a dynamic version of the static TEdit Include command, and is intended to facilitate the unbundling of document chapters and sections, while maintaining the ability to print the entire document or any portion of it. When an Include object is created, the user is prompted for a file name. An Include object can be enabled or disabled. If it is enabled, the object shows in the TEdit window as '@Include[MySubFile.TEdit]', and the indicated file will be included during a hardcopy operation. If it is disabled, then the object shows (both in the TEdit window and on hardcopy) as '@DoNotInclude[MySubFile.TEdit]'.

Middle-clicking on an Include object pops up a menu withthe following fields: "New File" (prompt for a new file name), "Edit File" (TEdit the Include file, or bring it to the top if is already being edited), "Enable" (include the file during hardcopy), and "Disable" (do not include the file during hardcopy).

Two caveats:

1) For best results, make an Include object the last thing in a paragraph, or put it in a paragraph of its own, and set the line and paragraph leadings to 0. The Include object forces a paragraph break right after the Include object during hardcopy, to prevent the looks of the paragraph containing the Include object to mask the looks of the first paragraph in the file being included.
2) A document containing Include objects is best hardcopied from a FileBrowser window, rather than through the hardcopy command on the TEdit window menu. It will work properly either way, but it's a bit unnerving to watch TEdit trying to reflect on the display the inclusion ofone or more files before hardcopy and the removal of the included files after hardcopy.

Horizontal Rule [Document Object]

This provides a more user-friendly interface to the HRULE package (which is automatically loaded by the DOC-OBJECTS package). Upon selecting it a numberpad is brought up repeatedly, with which the user can indicate the thickness of alternating black and white lines. The resulting HRule object is inserted in the document whenever the numberpad is aborted or returns 0. The DOC-OBJECTS package also modifies the HRule object such that it can be edited: clicking the middle button over an HRule object brings up a structure editor (such as SEdit) on a list containing the thicknesses of the lines composing the HRule. This list can be altered in any way, as long as the editor returns another list of numbers (presumably of odd length).

Eval'd Form [Document Object]

Selecting this object causes a type-in window to pop up. The value of the form typed in is assumed to be an image object. This is what TEdit used to do on typing CTRL-O. For TEdit's purpose, an image object is a Lisp value of type IMAGEOBJ, BITMAP, STRINGP, LITATOM, or REGION. The latter is assumed to refer to a region of the screen.

Screen Snap [Document Object]

Selecting this object prompts for a region of the screen. A bitmap containing a copy of the given region of the screen is inserted in the document. This is equivalent to clicking the right button in the display's background while holding the SHIFT key down.

Extending the Document Objects interface

DocObjectsMenuCommands [Variable]

This variable contains a list of menu items which are displayed in the Document Objects pop-up menu. It is analogous to the variable BackgroundMenuCommands (*cf.* IRM, Section 28.8). The Lisp form in each item is assumed to evaluate to an image object (as defined under Eval'd Form described above).

DocObjectsMenu [Variable]

This variable caches the Document Objects menu. Set it to NIL whenever you alter the variable DocObjectsMenuCommands.

DocObjectsMenuFont [Variable]

This variable contains a font descriptor which is used for displaying the items in the Document Objects menu. The initial value is (FONTCREATE '(MODERN 12 BOLD)). Set the variable DocObjectsMenu to NIL whenever you alter DocObjectsMenuFont.

(DOCOBJ-STRING-IMAGEBOX STRING IMAGESTREAM) [Function]

A useful function for Document Objects that wish to display as a string of characters (such as a TimeStamp). The Document Object's IMAGEBOXFN can call this function to obtain an image box with the TEdit Looks that apply to the Document Object taken into account.

(DOCOBJ-WAIT-MOUSE WINDOWSTREAM) [Function]

A useful function for Document Objects that wish to assure that their buttoneventfn takes action only if the mouse buttons were let up within the Object's region (i.e., the clipping region of the Object's window stream). It returns T when the mouse buttons go up within the region, or NIL when the mouse moves out of the region while a button is still down.

Future predefined Document Objects

Watch this space for objects such as Index & Index Entry, Citation & Bibliography, ...

DONZ

The *Excruciatingly* User Friendly Environment

By: Jeff Shrager et al.

Checked out for Medley by Larry Masinter (Masinter.PA@Xerox.COM)

Files: DONZ. DONZ.dfasl DONZ.TEDIT

Description:

Loading DONZ starts a background process (DONZ.RUN) which causes your ICONS to become "user friendly". Telling you what this means would spoil all the fun of discovery.

Customizing DONZ:

The delay between activations of DONZ is done by (DISMISS DONZ.DELAY). Its value defaults to 5000 (5 seconds). When DONZ wakes up, it trys out one window, selected at random, from all the windows in (OPENWINDOWS). If the selected window is not an ICON, nothing happens and DONZ wakes up again in 5 more seconds. If that window is an ICON, then DONZ tries to find a message for it as described below. This method results in DONZ's "friendliness rate" running approximately in proportion to the ratio of icons to opened windows on your screen. Thus, if you are doing real work, DONZ won't bug you, but if you just have a screen full of icons, DONZ will be *exceedingly friendly*.

The list DONZ.TEST.MESSAGE.ALIST has the form:

(frob1 frob2 frob3...)

where each frob is of the form:

(testfn msg1 msg2 msg3...)

TESTFN will be called with one argument: the *MAIN* window with which this icon is associated. That is, for instance, the TEDIT window that the icon will exand to...NOT the icon window. TESTFN should decide whether or not this window is of a type that it will handle, and return T or NIL as appropriate. When one of the TESTFNs returns T, one of the messages in the tail of the associated frob list will be selected at random and displayed...in the way that they are displayed...you'll see! If none of the frobs accepts responsibility for the present icon, there are a few default messages built into DONZ.

The msgs should be lists containing individual words (appropriately capitalized) so that .PARA in the PRINTOUT can do the appropriate word division.

Notes:

DONZ has to be killed via the PSW. Be gentle.

DONZ continues to run during screen idle. This results in fairly funny theatrics on the part of your icons.

The display is done with PRINOUT which does some bogusness to indent, erasing some of the good parts of the window. This is slightly messy, but otherwise innocuous.

Anyone who can guess the origin of the name of this package belongs on the East Coast.

Acknowledgements:

Thanks to Ros Chast for originating the idea, Mike Kazar and Dave Nichols for their original implementation of DONZ at CMU.

DORADOCOLOR

Maintained By: Frank Shih (Shih.envos@Xerox.com)

INTERNAL

Uses: COLOR

This document last edited on 8-Nov-88.

INTRODUCTION

This package is the Xerox Lisp software driver for the Dorado (Xerox 1132) color display.

NECESSARY HARDWARE

You need a Xerox 1132 with color card and a third party color display. Please contact your Xerox representative for details concerning acquiring and setting up all the required hardware. Some notes on configuring the Conrac color monitor are given at the end of this document.

Assuming you have all the hardware you need, turn it all on. This means

(1) Your 1132 is running Xerox LISP.

(2) Your 1132 has an 1132 color card installed.

(3) Your color display is plugged in and powered on.

(4) Three cables for red, green, and blue signal connect the 1132 color card to the color display.

Any reconnections should be made while your 1132 is off. Until you issue some software commands, a black color display is normal.

DORADOCOLOR SOFTWARE

The DORADOCOLOR package provides the machine dependent portion of software that is needed to drive your color display assuming you are using an 1132 with 1132 color card. Other than LOADing the DORADOCOLOR package and turning the DORADOCOLOR package on using the function COLORDISPLAY, all additional functionality is provided by and documented with the COLOR package. There are no DORADOCOLOR functions that the user needs to call directly. The user calls functions described in the COLOR documentation. A single exception to these comments is a global variable \DORADOCOLOR.LEFTMARGIN which controls the period of time the display controller should wait before turning on the color guns. \DORADOCOLOR.LEFTMARGIN is normally set to 56; if this value causes odd results with your monitor, try setting \DORADOCOLOR.LEFTMARGIN a little higher or lower and reinitializing the display. (You can reinitialize the display by calling COLORDISPLAY twice in succession).

Once your hardware is on, you can proceed to issue COLOR commands to your hardware. You should have the DORADOCOLOR package already LOADed from your LIBRARY directory. That is, you've already done something like (FILESLOAD DORADOCOLOR). At this point it may be convenient to follow this documentation along with the documentation for COLOR in the Lisp Library Packages Manual. If you now type

(COLORDISPLAY 'ON 'DORADOCOLOR)

your display will now change from total black to a color test pattern with horizontal and vertical stripes. The sequence of events is that there should be a noticable flicker on your color display, followed by a white wall covering the color display, followed by the painting of this white wall with horizontal and vertical strpes of color woven together. There are now some simple tests you can do to satisfy yourself that your hardware is working. Here is a small list of things to try:

```
(SETQ CSBM (COLORSCREENBITMAP))

(BLTSHADE 'WHITE CSBM)

(BLTSHADE 'RED CSBM)

(BLTSHADE 'GREEN CSBM)

(BLTSHADE 'BLUE CSBM)

(SETQ DS (DSPCREATE CSBM))

(DRAWLINE 0 0 500 500 10 'REPLACE DS 'YELLOW)

(DRAWLINE 500 0 0 500 10 'REPLACE DS 'CYAN)
```

Assuming all has gone well to this point, you should now be able to try all the functions described in the COLOR package documentation. The COLORDEMO package is a good source of test programs to try — (IL:LOAD 'COLORDEMO.LCOM) to get this package. Both COLOR and COLORDEMO documentation are in your LispUsers' Manual.

KNOWN BUGS

As of 11/88, there are several known bugs with the color code. Dragging the mouse off the right hand edge of the display appears to hang the Dorado. Also, color fonts do not seem to work (probably because in Medley AC fonts are unrotated using pilotbbt) .

APPENDIX - THE CONRAC COLOR MONITOR

This section describes configuring the Conrac color monitor (model 7211C19) for use with the Dorado**.**

Back Panel - Connections need to be made as follows: each color cable (red, green, blue) should be connected to the color-corresponding IN terminal. The black cable should be connected to the SYNC/HDRIVE IN terminal. The rocker switches next to the connectors should be set to 75 ohms.

Front Panel - The BRIGHT and CONTrast knobs push in to configure to their preset settings, pull out for adjustment. The SCREEN buttons should be toggled out, the CHANNEL button should be toggled out.

Internal Switches - If after connecting the monitor and starting up the color software, the display does not appear to be in sync, you may need to check the internal settings inside the monitor. To do so, you will need to remove the cover and several internal metal panels (which may be screwed to the bottom of the monitor as well). Viewed from the front of the monitor, on the left hand side is the Scan Board, on the right hand side is the Video Board. Adjustments should be made with the power off, exercise caution!

On the Scan Board, there is a Scan Rate Jumper (pins 52-61) which controls the scan rate (different resolutions). This jumper should be in the bottom position (pins 52-59), specifying low resolution (525 lines).

Below it to the left are three potentiometers, the lower one (R126) controls the horizontal hold for low resolution. Other "pots" of interest control pincushion (R65), width (R78), height (R4), v-hold (R2), v-center (R24), h-center (R90).

On the Video Board, there is a Sync Jumper (pins 35-37) which controls the sync rate. This jumper should be in the upper position (pins 35-36).

Further details can be found in the Conrac 7211 Manual.

DSPSCALE

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

DSPSCALE allows a program to output to different types of streams (display, Interpress, etc.) without corrections for scaling. This module provides self-scaling graphics through two different methods: a virtual self-scaling *image stream* that overlays a regular image stream and/or new versions of the various image stream graphic manipulation functions (DRAWLINE, DSPTOPMARGIN, etc.). The goal of both methods is to make it possible to modify the normal scaling factor of an image stream without modification to the program generating the output.

VIRTUAL SELF-SCALING IMAGE STREAM

This module implements a virtual image stream type, called SCALED, which is used to overlay any other image stream and provide automatic scaling to the natural scale of the image stream or any user selected scaling factor. The function OPENIMAGESTREAM is used to overlay a scaled image stream over a regular one. For example, the following will open a scaled image stream on top of an Interpress image stream:

```
(OPENIMAGESTREAM (OPENIMAGESTREAM 'TEST.IP 'INTERPRESS) 'SCALED)
```

The only difference between the virtual stream and a normal image stream is that the SCALE argument to the DSPSCALE function is active and can be used to change the scale of the stream (multiplying the scale specified by the standard scaling factor of the stream).

SELF-SCALING GRAPHICS FUNCTIONS

As an alternative to the self-scaling image stream, self scaling versions of the various graphics functions are provided. For most of the graphic functions, self-scaling versions have been defined which have an **!** (exclamation point) at the end of their name (eg. DRAWLINE vs. DRAWLINE**!**):

```
   CENTERPRINTINREGION!       DRAWELLIPSE!          DSPSCALE!
   CHARWIDTH!                 DRAWLINE!               RELDRAWTO!
   CHARWIDTHY!                DRAWPOINT!              RELMOVETO!
   CURSORPOSITION!            DRAWPOLYGON!            SCALEDBITBLT!
   BITBLT!                    DRAWTO!                 STRINGREGION!
   BITMAPBIT!                 FILLCIRCLE!             STRINGWIDTH!
   BLTSHADE!                  FILLPOLYGON!          DSPSPACEFACTOR!
DSPBACKUP!                    FONTPROP!             DSPTOPMARGIN!
DSPBOTTOMMARGIN!              GETPOSITION!          DSPXOFFSET!
DSPCLIPPINGREGION!         DSPLEFTMARGIN!           DSPXPOSITION!
   DRAWARC!                DSPLINEFEED!             DSPYOFFSET!
   DRAWBETWEEN!               MOVETO!               DSPYPOSITION!
   DRAWCIRCLE!                MOVETOUPPERLEFT!
   DRAWCURVE!              DSPRIGHTMARGIN!
```

The set includes both output *and* input functions since it is necessary when getting, for example, a mouse position, to unscale the position to put it back into the program's virtual coordinate system. By default, these functions can be used directly in place of their non-**!** counterparts and they will automatically scale their arguments to the DSPSCALE of the output stream. Some of the above functions are not identical with their non-**!** counterparts, as explained below:

(DSPSCALE! *SCALE STREAM*) [Function]

In this version of DSPSCALE, the *SCALE* argument is active and will multiply *STREAM*'s normal scaling factor. If you have a program that draws a circle, for example, to a window using the appropriate **!** functions, you can cause it to draw a different size circle (larger or smaller) by using DSPSCALE! to change the scaling factor of the window without touching the source program.

(CHARWIDTH! *CHARCODE FONT* ***STREAM***) [Function]

(CHARWIDTHY! *CHARCODE FONT* ***STREAM***) [Function]

(FONTPROP! *FONT PROP* ***STREAM***) [Function]

(STRINGWIDTH! *STR FONT FLG RDTBL* ***STREAM***) [Function]

All of the above functions have one extra argument (as compared to their non-**!** equivalents) which is the *STREAM* in question. This is necessary to do the scaling calculations.

The module also defines a couple of new stream manipulation functions:

(DSPTRANSLATE! *Tx Ty STREAM*) or (DSPTRANSLATE! *POSITION STREAM*) [Function]

Defines the amount of X and Y translation that should be added to graphic operations to *STREAM*. Similar to DSPTRANSLATE but works even if the image stream does not have an IMTRANSLATE method. The second form of the arguments is for backward (Koto) compatibility.

(DSPUNITS! *UNITS STREAM*) [Function]

Essentially the inverse of DSPSCALE!, this function lets you set how many *UNITS* (pixels or whatever) the source program generates for each unit pixel on the output stream (multiplied by the output stream's default scaling).

It is possible to use both the virtual image stream and the self-scaling graphics functions together as long as the self-scaling graphics functions are applied to the real stream, not the virtual one.

Lisp Data Type Scaling Functions

The routines below are used by the **!** functions and the virtual image stream for scaling numbers, positions, regions and other data types and are useful for defining other self-scaling functions:

(DSPSCALE.BRUSH *BRUSH STREAM*) [Function]
(DSPSCALE.DASHING *DASHING STREAM*) [Function]
(DSPSCALE.POINTS *KNOTS STREAM*) [Function]
(DSPSCALE.REGION *REGION STREAM [SmashRegion]*) [Function]
(DSPSCALE.NUMBER *NUMBER STREAM*) [Function]
(DSPSCALE.POSITION *POSITION STREAM [SmashPosition]*) [Function]
(DSPSCALE.XPOSITION *NUMBER STREAM*) [Function]
(DSPSCALE.YPOSITION *NUMBER STREAM*) [Function]
(DSPSCALE.WIDTH *WIDTH STREAM*) [Function]

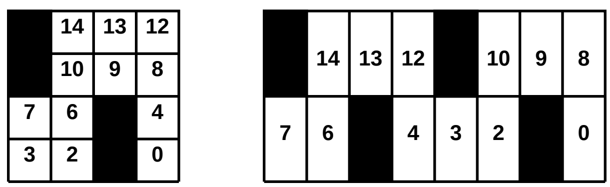
(DSPUNSCALE.REGION *REGION STREAM [SmashRegion]*) [Function]
(DSPUNSCALE.POSITION *POSITION STREAM [SmashPosition]*) [Function]
(DSPUNSCALE.NUMBER *NUMBER STREAM [OFFSET]*) [Function]
(DSPUNSCALE.XPOSITION *NUMBER STREAM*) [Macro]
(DSPUNSCALE.YPOSITION *NUMBER STREAM*) [Macro]

EDITBG

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

EDITBG is a tool for editing both the background and background border shades. The functions CHANGEBACKGROUND and CHANGEBACKGROUNDBORDER both take a shade argument but the shade is interpreted differently. A normal black & white shade consists of 16 pixels (see EDITSHADE in the Interlisp Reference Manual) as does the border shade, which covers twice the area. The normal shade has 4 x 4 pixels but the border shade has 2 x 8 pixels where the pixels are twice as tall. WHITESHADE and BLACKSHADE appear the same for both, as does the standard background shade (shown below) but arbitrary shades do not appear the same.

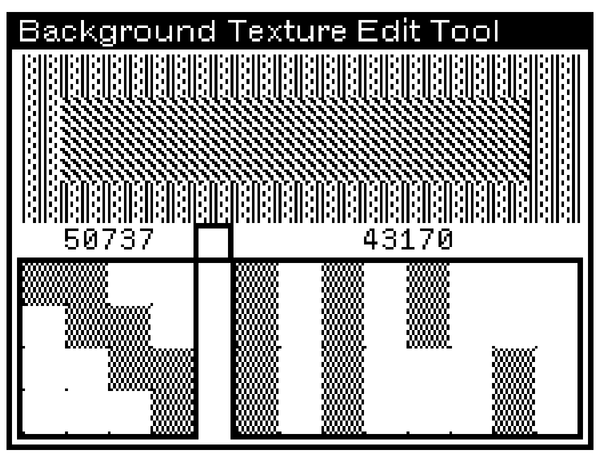
Background **Background Border**



34850 = 2^15 + 2^11 + 2^5 + 2^1

(EDITBACKGROUND) [Function]

Brings up an edit tool (also available from the background menu) which lets you edit both a normal shade and a border shade and see how they combine:



The bottom half of the window has a background texture editor on the left and a border texture editor on the right. The top half of the window shows the background texture within the border texture as it would appear on the screen. Buttoning the small box in the center of the window will change the background and border textures on the screen to those displayed.

EDITKEYS

By: Larry Masinter (Masinter.pa@Xerox.com)

EDITKEYS provides a set of "logical" keys corresponding to the Dandelion (1108) function keys. This allows 1132 users to take advantage of interfaces designed for the 1108 keyboard. Calling (BUILDFNKEYS) builds a window with 8 keys like the one below:

These keys can be "pressed" by bugging the mouse (left or middle mouse button) inside the key's image. The effect of pressing one of these keys is to generate the same character code as the key generates on the 1108. The state of the shift keys (at the time the mouse is let up) are taken into consideration, but interfaces that use KEYDOWNP are not affected.

EQUATIONEDITOR

PROGRAMMER'S GUIDE

By: Tad Hogg (Hogg.pa@Xerox.com)

DESCRIPTION

This document describes how to define new kinds of equations to be used with the equation editor in TEdit, and how to construct equation image objects by function calls.

User Switches:

The global variable *EquationDefaultSelectionFn* specifies the default function to be called when an equation is selected with the middle mouse button. The initial value, EQIO.DefaultSelectFn, allows the user to select a piece of the equation by selecting from a menu.

The global variable *UnknownEquationData* is a formated string to use for displaying equations whose types are not defined.

The global variable *EquationInfo* is used to record all currently defined equation types. The specification information can be obtained by calling

(EQIO.GetInfo *type info*) [Function]

which returns the specified info for equations of the given type. Any of the PROPS mentioned below for EQIO.AddType, or *formFn* or *numPieces*, can be used as a value for *info* to get the corresponding data for this type of equation. Example: (EQIO.GetInfo 'fraction 'numPieces) returns the number of pieces in a fraction.

Individual specification items of an existing equation type can be modified using

(EQIO.SetInfo *type info newValue*) [Function]

although the caller must be sure that any new information is consistent with the remaining properties. For example, (EQIO.SetInfo 'fraction 'menuLabel myLabel) will make the value of myLabel be used for fractions in the equation type menu.

Defining new kinds of equations:

This module allows new types of equations to be defined. An equation consists of some number of pieces of text and, perhaps, some extra symbols or lines. A method for computing the relative position of the various pieces must be specified when new equation types are used. The pieces of equations consist of "formatted strings" which allow font information to be associated with strings and can also include image objects (which thus allows equations to contain other equations). Formatted strings are described below. Additional properties can be specified to determine the particular behavior of the new equation.

Additionally, a function can be provided to allow equations of the new type to be created under program control. Typically these are named EQ.Make.xxx where xxx = atom specifying the equation type.

Specifically, a new equation type (or a new definition for an existing type) is created by calling

(EQIO.AddType *type formFn numPieces PROPS*) [Function]

where

- *type* is an atom identifying the equation type (e.g. fraction)
- *formFn* is the name of a function, described in detail below, which specifies the relative location of the equation pieces and, if requested, draws any extra lines or symbols required by the equation that are not included in any of its pieces.The formFn can also specify the selection region to be used for each piece of the equation, i.e. that region of the equation within which the left mouse button can be used to select the piece.
- *numPieces* is the number of parts the equation has (e.g. a fraction has two parts: numerator and denominator). If the equation has a variable number of pieces, then numPieces is the default initial value.
- *PROPS* is a prop list of optional properties for the equation which are described below.

The equation form function:

The form function is called with arguments (eqnObj imageStream draw?) and specifies the size of the entire equation and the location of each piece with respect to the lower left corner of the box. Furthermore, if *draw?* is non-NIL, it draws any extra lines or symbols required by the equation that are not included in any of its pieces. (If *draw?* is NIL, nothing should be drawn -- the function is being called only to determine how big the equation is and the relative location of its parts.) The formFn can also specify the selection region to be used for each part of the equation.

For example, a fraction has two parts (numerator and denominator) and a single extra line between them. In this case the formFn draws the line and specifies the location of the numerator and denominator.

Specifically, the formFn should return an equation specification created by a call to

(EQIO.MakeSpec *box dataSpecList*) [Function]

where *box* is an IMAGEBOX which specifies the size of the equation; and *dataSpecList* is a list containing a piece specification for each piece of the equation. A piece specification is created by a call to

(EQIO.MakeDataSpec *pos selectRegion*) [Function]

where *pos* is the position, relative to the lower left corner of the box, where this piece is to be displayed and *selectRegion* gives the selection region to use for this piece (relative to the l.l. corner of the equation box). If *selectRegion* is NIL, then the region within which the piece is displayed will be used as the selection region. Normally the selectRegion should include the display region inside it, and is intended to allow selection regions to be larger than just the display region. Note that *pos* specifies where the stream should be positioned before displaying the formatted string defining the contents of the piece.

The ATTACHEDBOX routines, described below, may be useful for constructing the form functions of new equations since it provides functions to position various regions so that they do not overlap.

Equation type properties:

The following properties can be specified for any equation type when it is defined with EQIO.AddType. Note that all equations of a given type have the same values for these properties.

- *changeFn* A function with argument (eqnObj) called when the number of pieces in a variable piece equation is changed. It is meant to allow any specific properties such as saved menus to be adjusted to reflect the change.
- *initialData* A way to specify the text to be used when equations of this type are created. It can be either the name of a function or a list. If it is a list, then it should contain a single integer for each piece of the equation. This number specifies how the font size for that piece should be changed relative to the initial font size set in TEdit. For example, 0 means use the default font, +1 means use the next bigger font, -2 means use a font two sizes smaller, etc. Missing values default to zero, i.e. the pieces are initially set in the normal size font. The actual fonts used are specified by the array EquationFontSpecs and any request for a font larger (smaller) than the largest (smallest) available in the array defaults to using the largest (smallest). In this case, the initial text will be a single blank.

If initialData is a function, it is called with arguments (initialFontSpec type numPieces dataList) when a new equation of this type is created. It should return a list of formatted strings, with each item in the list to be used for the corresponding piece of the equation. The argument initialFontSpec will specify the current font when the equation is added; and numPieces will be the number of pieces in the equation. dataList, if non-NIL, is a list of items to use for each of the pieces of the equation. The items can be either format strings, or just a string in which case the initialData function should attach an appropriate font.

- *initialPropFn* A function with argument (type) called when a new equation of this type is created. It returns a prop list to be used as properties for this equation. These values are added to (and override) any props given in objectProps. For example, this allows the user to specify the number of rows and columns in a new matrix. If the number of pieces is to be specified, it should be returned as the value of the numPieces prop, and the equation type should allow a variable number of pieces.
- *makeFn* A function which constructs an image obj for this type of equation from its arguments. Generally this will just provide a convenient way of calling EQN.Make.
- *menuLabel* A label to use for this type in the equation type menu displayed after selecting Equation in the main TEdit menu. If not given, the type atom is used as the label.
- *objectProps* A prop list of properties and their initial values that are needed for this equation. These properties can be used by the formFn to lay out the equation and will typically be modified by the wholeEditFn.
- *pieceNames* A list of names of the pieces of the equation. This is used by the default middle button selection function to provide a menu of choices. E.g. ("numerator" "denominator") for a fraction. If this property is not specified, then the default menu will just contain the numbers of the pieces. This is generally meant for equations with a fixed number of pieces.
- *specialSelectFn* A function with argument (eqnObj) to be called when the equation is selected with the middle button instead of the default action. It should return the number of the piece selected, or NIL if no piece of the equation is selected. The selected piece, if any, will then be edited.
- *wholeEditFn* A function with arguments (eqnObj window button) called when the entire equation, rather than a single piece, is selected. This can be used to change global properties of

the equation and should return non-NIL if the object is modified, NIL otherwise. *window* is the window which contains the equation and *button* is the mouse button used to select it.

- *variable?* Non-NIL to indicate this equation type allows a variable number of pieces.

Equation data functions:

The functions used with new equation types should make use of the routines provided for formatted strings as well as the following functions when using the various equation data:

- (**EQIO.EqnType** eqnObj) returns the atom specifying the kind of equation *eqnObj* is.
- (**EQIO.EqnDataList** eqnObj) returns the list of formatted strings which specify the pieces of the equation.
- (**EQIO.SetDataList** eqnObj newDataList) replaces the data list (i.e. the list of formatted strings corresponding to each piece of the equation) with newDataList. The caller must update the number of pieces in the equation appropriately. This is useful, for instance, when the wholeEditFn has made major changes in the equation.
- (**EQIO.EqnData** eqnObj piece#) returns the formatted string corresponding the the piece of *eqnObj* specified by piece#.
- (**EQIO.EqnProperty** eqnObj prop {newValue}) returns the current value of the specified property of *eqnObj* if *newValue* is not present, otherwise sets the specified property to the value of *newValue* even if it is NIL. This can be used to associate arbitrary properties with individual equations. Currently, the following properties are used by the equation editor and should not be used for other purposes:
  - *fontSpec* a specification of the current font at the time the equation was created
  - *numPieces* the current number of pieces in a variable-piece equation
  - *selectionMenu* the current default middle-button selection menu for a variable-piece equation

When equations are copied, any data items that are not atoms, strings or lists are set to NIL. These items are also PRIN2'ed on files. Thus other data types should only be used to cache values (e.g. menus) that can be recomputed if necessary from the other properties.

- (**EQIO.NumPieces** eqnObj {newValue}) returns the current number of pieces in *eqnObj* if *newValue* is not present. Otherwise if *eqnObj* is a variable-piece equation, it sets the number of pieces to *newValue* and adjusts any necessary properties by calling the equation's changeFn.

Additionally, properties can be associated with the equation type itself by use of

- (**EQIO.TypeProp** type prop {newValue}) which gets or sets the property *prop* for equation *type*. This can be used to save properties that are the same for all equations of a given type (e.g. selection menus for equations with a fixed number of pieces).

Equation image objects can be created by a program (and then, for example, inserted into TEdit) by use of

- (**EQN.Make** type dataList fontSpec PROPS) which makes a *type* equation whose arguments are format strings contained in *dataList*. *fontSpec* is an initial font specification and *PROPS* is a prop list of equation properties which should include *numPieces* for equations with a variable number of pieces.

FORMATSTRINGS

The basic components of equations are represented as formatstrings which allow fonts and super/subscripting to be associated with strings and can also include image objects. A format string is a list of items, each of which is either an imageobject or a list giving a font specification, a string and an optional shift specifying the number of points to move up when displaying the string. The following functions can be used to create and display format strings as well as insert and extract them from TEXTSTREAMs. All formatstrings must be on a single line.

Functions:

For creating and accessing format strings:

(FS.MakeItem *fontSpec string shift*) [Function]

creates a formatstring item from *fontSpec*, a font specification such as (Gacha 10), *string* and *shift*. The string can be null.

(FS.ItemFont *item*) [Function]

returns the font associated with *item*, or NIL if *item* is an imageobject.

(FS.ItemValue *item*) [Function]

returns *item* if it is an imageobject, otherwise its associated string.

(FS.ItemShift *item*) [Function]

returns the shift associated with *item*.

For inserting and extracting from TEXTSTREAMs:

(FS.Extract *stream*) [Function]

returns a format string created from the text in the TEXTSTREAM stream. Any unallowed characters (as determined by FS.AllowedChar) in the stream are ignored. Note that any format information other than character fonts, super/subscripting and image objects is discarded. The file pointer associated with the stream is modified.

(FS.Insert *data stream*) [Function]

inserts the format string *data* at the current location in TEXTSTREAM stream.

For displaying and manipulating the format strings:

(FS.Box *data imageStream*) [Function]

returns an IMAGEBOX specifying the size of the format string *data* on *imageStream*.

(FS.Copy *data*) [Function]

returns a copy of the format string *data*.

(FS.Display *data imageStream invert?*) [Function]

displays the format string *data* on *imageStream*. If *invert?* is non-NIL, the display is inverted.

(FS.Get *fileStream*) [Function]

reads a formatstring, or list of formatstrings, from the current location on *fileStream*.

(FS.Put *data fileStream*) [Function]

prints the formatstring (or list of formatstrings) *data* to *fileStream*.

Additional functions:

(FS.AllowedChar *charcode*) [Function]

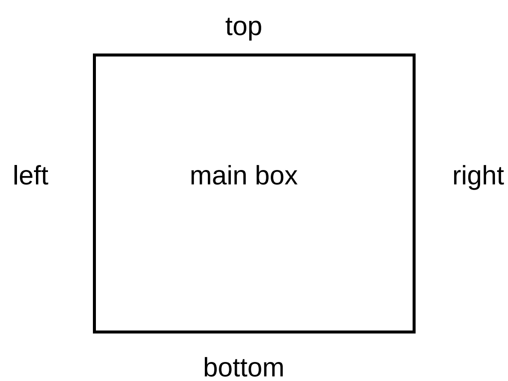
returns non-NIL if *charcode* is allowed in formatstrings.

(FS.RealStringP *item nullOK*) [Function]

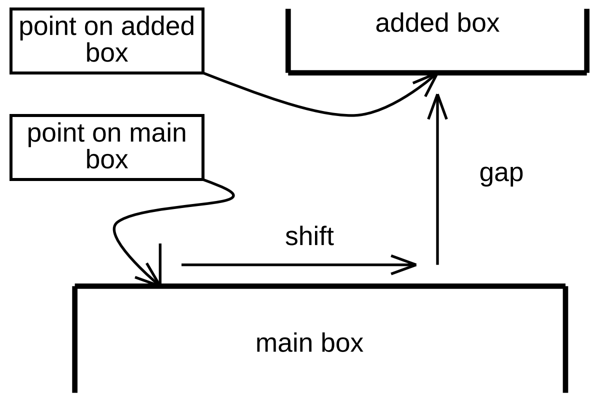
returns non-NIL if *item*'s value is a string (rather than an imageobject) and either *nullOK* is non-NIL or the string is not the nul string.

ATTACHED BOXES

The following functions place image boxes in specific locations with respect to a main box so that the added boxes won't overlap. The desired position of a new box is specified by the side of the main box to place it next to and the position of a point on the side of the new box with respect to a point on the side of the main box. The placed regions are specified with respect to the lower left corner of the main box. Sides are specified by one of the atoms *top*, *bottom*, *left* or *right* and are with respect to the main box.



The position of the added box is specified relative to some side of the main box. Specifically, the location of a reference point on the near side of the added box, the addPt, is given with respect to a reference point on the side of the main box, the mainPt. The possible points along the side are specified by one of the atoms *low*, *high*, *center* or *display* corresponding to the corner nearest the lower left corner of the box, the corner farthest from the l.l. corner of the box, the center of the side, and the display point of the image box respectively. The location of the addPt with respect to the mainPt is specified by a distance along the side, the shift, and a distance perpendicular to the side, the gap. These distances can be positive or negative. A negative value for the gap will cause the added box to overlap the main box. All boxes are assumed to have no kerning (i.e. XKERN field is zero).



Functions:

For creating and accessing format strings:

(AB.PositionRegion *mainBox addedRegions side mainPt addBox addPt gap shift clear*) [Function]

Positions *addBox* with respect to *mainBox* avoiding overlap with previously added regions. The parameters are: *mainBox* is an image box specifying the main box; *addedRegions* is a list of regions (measured with respect to the lower left corner of *mainBox*) that have already been placed next to the main box; *side* is the side (one of *top*, *bottom*, *left* or *right*) of the main box next to which the new box should be placed; *mainPt* is the reference point along the side of the main box (one of *low*, *high*, *center* or *display*); *addBox* is an image box specifying the box to be placed; *addPt* is the reference point along the side of the added box; *gap* and *shift* specify the relative positions of the reference points; and *clear* is the minimum (nonnegative) distance that the new box is allowed to be from any of the previously added regions (NIL defaults to zero which prevents any overlap of the added regions).

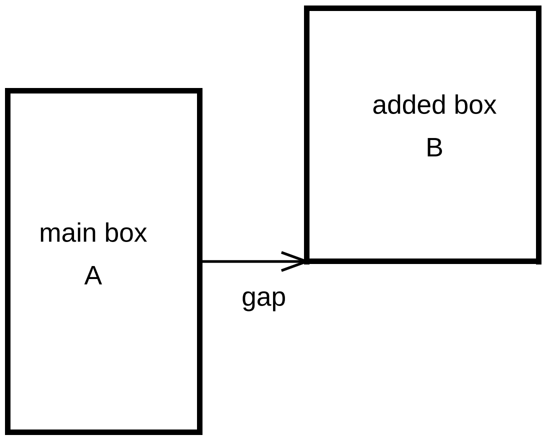
The function returns a list of the form (*region newAddedRegions*) where *region* is the region, w.r.t. the lower left corner of *mainBox*, where *addBox* was placed and *newAddedRegions* is the list of added regions updated to include this newly placed region. If the specified location of *addBox* causes it to be within a distance *clear* of any of the regions in *addedRegions*, the box is moved away from the main box in a direction perpendicular to the side (i.e. the gap is increased) until it is far enough from the previous regions.

(AB.Position2Regions *mainBox addedRegions side highBox highPt lowBox lowPt highGap lowGap highShift lowShift clear*) [Function]

This function places two boxes next to the same side of *mainBox*. If the two new boxes are within a distance *clear* of each other, they are moved apart in a direction parallel to the side next to which they are placed so that the distance each box moves is proportional to its size. Then these regions are individually checked for being too close to previously added regions and, if necessary, are moved away from the main box (i.e. perpendicular to the side). The function returns a list of the form (*highRegion lowRegion newAddedRegions*).

Example:

To place box B to the right of box A such that the low point of the near side of box B is a distance *gap* from the center of the right side of A, i.e.



use (AB.PositionRegion *A addedRegions* '*right* '*center B* '*low gap* 0) where *addedRegions* is a list of previously added regions which should not overlap *B*.

EQUATIONS

By: Tad Hogg (Hogg.pa@Xerox.com)

DESCRIPTION

This module provides interactive editing of mathematical equations within TEdit. An equation consists of a number of pieces of text and possibly one or more special symbols. For many purposes, such as deletion and copy selection, equations behave as a single (large) character in TEdit. Operations on their pieces are described below.

To load:

The equation editor and a standard set of equation types is obtained by loading EQUATIONS.LCOM.

To use:

This section describes the procedures by which equations can be inserted into documents and their pieces modified.

Adding an equation to a TEdit document:

To add an equation, first move the caret to the desired insertion point and then select "Equation" from the main TEdit menu (obtained by holding down the middle button in the window’s title bar). This will display a menu of known equation types. Selecting one of these will insert the corresponding equation into the document at the current location of the caret. To abort the insertion, click outside the menu. Once the equation is inserted, a subeditor will be created for each of the equation pieces, one at a time. This can be used to fill in the various pieces of the equation and its use is described below. Some equation types will prompt for additional information before inserting the new equation (e.g. inserting a matrix will prompt for the desired number of rows and columns).

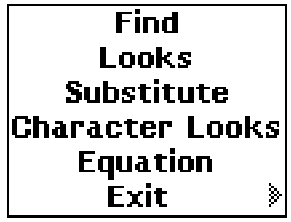
Editing a currently existing equation:

In order to modify a piece of an existing equation (e.g. the numerator of a fraction), the piece must be selected with the mouse in one of two ways. First, you can point at the piece in the displayed equation and press the left mouse button. Alternatively, pointing at the equation and pressing the middle button will display a menu from which the desired piece can be selected. This is useful for selecting pieces that are too small to conveniently point at with the mouse. In either case, if a piece is selected, a subeditor will start on that piece. In addition, some equation types may also allow changes to global properties when no specific piece is selected (e.g. changing the number of rows in a matrix).

Using the equation piece subeditor:

The subeditor is attached to the bottom of the main edit window and allows individual pieces of the equation to be modified with normal TEdit operations. While a subeditor is active, the corresponding piece of the equation in the main window is inverted. Since the text in equation pieces must be on a single line, the subeditor will not accept control characters such as carriage returns. Instead the edit window will flash when such characters are typed.

In the subeditor, the TEdit menu is modified to provide a limited set of TEdit commands as well as additional commands relevant to equations. The menu appears as



Selecting *Find*, *Looks*, *Substitute* or *Character Looks* invokes the corresponding TEdit action. Selecting *Equation* acts as described above and allows equations to be embedded inside other equations. *Exit* ends the subedit of the equation piece, updates the equation in the main editor and, if this is a newly inserted equation, automatically starts editing the next piece.

The *Exit* item also has three possible subitems which are used to exit from the equation editor and specify a desired follow up action. Specifically, *Next Piece* ends the edit of the current equation piece and creates a new editor on the next piece. In this context, the pieces of the equation are considered to form a circular list so that successive uses of the *Next Piece* option will edit each piece of the equation in turn. The second subitem, *Finish Eqn*, ends the current equation edit and does not continue with any other pieces of the equation. Finally, *Abort* ends the current edit without changing the equation.

When a subeditor is terminated, any TEdit looks or formatting other than character fonts and sub/superscripting are ignored.

The subeditor can also be terminated by the key normally used to advance to the next fill-in slot in TEdit (i.e. text of the form ">>...<<"). Specifically, if there are no remaining slots in the subeditor, using this key is equivalent to selecting *Exit* from the command menu described above. By default, this key is the middle-blank key on Dolphins and Dorados and the OPEN key on DLions.

User Switches:

The global variable *EquationFontSpecs* is an array of font specifications in order of increasing size which is used to determine initial fonts for the equation pieces. This can be modified if additional or different default fonts are desired.

The global variable *EQ.UseNSChars* determines the kind of characters to use when displaying equations that use special symbols (e.g. sum or product) on the screen. Specifically, if non-NIL then symbols from the NS character set are used, otherwise the Sigma 20 font is used. It is initially set to NIL.

When NS characters are used (on the screen when *EQ.UseNSChars* is non-NIL, or for Interpress), the global variable *EQ.NSChars* determines the particular NS characters to use . It is a property list of the form (TYPE1 ITEM1 ...) where TYPE is the kind of equation (e.g. SUM, PRODUCT, etc) and ITEM gives the font and character number to use, e.g. ((MODERN 30) 61301) for an INTEGRAL. This variable compensates for the lack of large symbols in various Interpress fonts.

The file EQUATIONPROGRAM.TEDIT describes how to define new kinds of equations, as well as how to create equations in a program with function calls.

Examples:

Examples of equations are given in the file EQUATIONEXAMPLES.TEDIT. The equation module must be loaded before reading this file.

Limitations:

- Equations that are larger than the available room in the current TEdit window will not be displayed.
- The text of each piece of an equation must be on a single line.
- All image objects inserted into equations, as well as the equations themselves, must not have any kerning, i.e. the XKERN field of all imagebox records must be zero.

Koto Incompatibility:

Due to a change in image object I/O, files containing equations written in Lyric/Medley may not be readable in Koto.

ETHERBOOT

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: Various microcode, germ and boot files.

ETHERBOOT is a Envos Lisp background network server process which allows Dandelions and/or Doves (other than the one the server is running on) to boot utility programs from the Ethernet (as an alternative to floppies). On a Dandelion, a 3, 4 or 6 boot from the maintenance panel initiates an Etherboot; on a Dove the boot icons are used (sometimes in combination with a number key):

Dandelion	Dove	Boot Type
0003	F3	Ethernet non-diagnostic boot of the Installer
0004	F7	Ethernet diagnostic boot of the Installer
0006	F3-1	Ethernet boot of experimental software

(ETHERBOOT *[LOGFILE]*) [Function]

To start the server, `(ADD.PROCESS '(ETHERBOOT))`. *LOGFILE* is an optional argument which should be an open stream to log transactions in.

BOOTFILEDIRECTORIES [Variable]

The boot files are searched for on the directories in this list which should point to the (possibly remote) directory where the boot files are kept, initially `'({CORE} {DSK})`. The server will not respond to requests for boot files that are not available.

(CACHE.BOOT.FILES *[TYPES]*) [Function]

Since Lisp can take longer to open a remote file than the timeout on some (simple) requests, this function can be used to copy some of the boot files listed in ETHERBOOTFILES to the BOOTFILECACHEDIRECTORY. *TYPES* defaults to those listed in BOOTFILECACHETYPES.

BOOTFILECACHEDIRECTORY [Variable]

The directory into which CACHE.BOOT.FILES copies boot files, initially `{CORE}`.

BOOTFILECACHTYPES [Variable]

The default types of files that CACHE.BOOT.FILES copies to the BOOTFILECACHEDIRECTORY, initialy `'(DB GERM)`.

BOOTFILEREQUESTTYPES [Variable]

An association list which contains the type numbers of the requests that the boot server handles along with a description of the request type and the function which handles it. Currently, the request types are *Simple* and *SPP*.

ETHERBOOTFILES [Variable]

The table of boot file numbers and names. Each entry consists of a description of the boot file, the name of the file and the file number (48 bit) by which the file is requested. Since the boot server is table driven, different boot files can be substituted. Initially, ETHERBOOTFILES contains:

```
(("Standard DLion Ethernet Initial Microcode"       EtherInitial.db              2852126720)
("Standard DLion Diagnostic Microcode"              MoonBoot.db                  2852126728)
("Standard DLion Mesa Microcode"                    Mesa.db                      2852126736)
("Standard DLion Germ"                              DLion.germ                   2852126744)
("Standard DLion Boot File"                         SimpleNetExecDLion.boot      2852126752)
("Standard DLion Diagnostics Boot File"         EIDiskDLion.boot             2852127232)
("Standard DLion Installer Boot File"               InstallerNSDLion.boot        2852127234)

("Alternate DLion Ethernet Initial Microcode"       EtherInitialAlt.db           2852126721)
("Alternate DLion Mesa Microcode"                   Mesa.db                      2852126738)
("Alternate DLion Germ"                             DLion.germ                   2852126746)
("Alternate DLion Boot File"                        InstallerNSDLion.boot        2852126754)

("Standard TriDLion Diagnostic Microcode"           Moonboot.db                  2852126729)
("Standard TriDLion Mesa Microcode"                 TridentRavenMesa.db          2852126737)
("Standard TriDLion Germ"                           TriDlion.germ                2852126745)
("Standard TriDLion Boot File"                      SimpleNetExecTriDlion.boot   2852126753)

("Alternate TriDLion Mesa Microcode"                TridentRavenMesa.db          2852126739)
("Alternate TriDLion Germ"                          TriDlion.germ                2852126747)
("Alternate TriDLion Boot File"                     InstallerNSTriDlion.boot     2852126753)

("Standard Dove Ethernet Initial Microcode"         EtherInitialDove.db          2852128768)
("Standard Dove Diagnostic Microcode"               MoonRise.db                  2852128776)
("Standard Dove Mesa Microcode"                     MesaDove.db                  2852128784)
("Standard Dove Germ"                               Dove.germ                    2852128792)
("Standard Dove Boot File"                          SimpleNetExecDove.boot       2852128800)

("Alternate Dove Ethernet Initial Microcode"        EtherInitialDove.db          2852128769)
("Alternate Dove Diagnostic Microcode"              MoonRise.db                  2852128777)
("Alternate Dove Mesa Microcode"                    MesaDove.db                  2852128785)
("Alternate Dove Germ"                              Dove.germ                    2852128793)
("Alternate Dove Boot File"                         InstallerNSDove.boot         2852128801)

("Dove Simple Net Exec"                             SimpleNetExecDove.boot       2852128824)
("Dove Configuration Utility"                   SysConfigOfflineDove.boot    2852128825)
("Dove Installer"                                   InstallerNSDove.boot         2852128826)
("Dove Diagnostics Utility"                         DiagDiskUtilDove.boot        2852128828)
("Dove Rigid Disk Diagnostics Utility"              DiagRDDove.boot              2852128829)
("Dove Ethernet Diagnostics Utility"                DiagEtherDove.boot           2852128830)
("Dove Keyboard & Display Diagnostics Utility"      KDMDove.boot                 2852128831))
```

The boot file numbers overlay the host number space so Dandelion/Dove boot file numbers begin at 25200000000 octal.

KNOWN PROBLEMS

- The server can only handle one connection at a time.
- Due to as yet unknown reasons, a Dandelion running the server is not able to service simple Dove requests; all other combinations should work.

EXAMINEDEFS

By:

Ron Kaplan

This document created in December 2021.

EXAMINEDEFS brings up side-by-side windows for comparison of the definitions of NAME as TYPE from sources SOURCE1 and SOURCE2.

(EXAMINEDEFS NAME TYPE SOURCE1 SOURCE2 TITLE1 TITLE2 REGION) [Function]

The kind of comparison is determined by the value of the variable EXAMINEWITH. If EXAMINEWITH is SEDIT, the different definitions are shown in sided-by-side SEDIT windows. This allows for examination--but not editing--of the definitions: The structures shown in SEDIT are copies of the definitions that GETDEF obtains from the sources and so any changes in either SEDIT will have no lasting effect. A separate SEDIT session with a particular definition (maybe different from either of the sources) can be used in parallell for edits guided by the separate examination. Also, currently, the particular locations of differences in the two definitions are not highlighted by SEDIT selections.

If EXAMINEWITH is COMPARETEXT (the initial value), then the definitions are printed in read-only TEDIT windows (as per PF-TEDIT) and compared with COMPARETEXT. The COMPARETEXT browser makes it easy to iterate from difference to difference.

If SOURCE1 or SOURCE2 is a list and not a GETDEF source specification, then a copy of that structure will be taken as the definition to be shown.

If REGION is a region, it is used as the initial suggestion for the constellation region that side-by-side SEDIT windows will share. (COMPARETEXT windows are determined by the COMPARETEXT protocols.)

TITLE1 and TITLE2 if provided are used to override the default titles of the examination windows. The windows are attached, in that they move, reshape, and close together, but scroll independently.

A second function is provided to bring up side-by-side TEDIT-SEE windows as an aid in examining the differences between files. The optional REGION again specifies a constellation region that the TEDIT's for the two files will share.

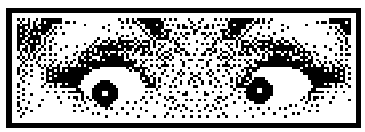
(EXAMINEFILES FILE1 FILE2 TITLE1 TITLE2 REGION) [Function]

EYECON

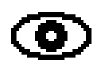
By: Bob Bane (Bane.envos@Xerox.com)

DESCRIPTION

EYECON creates a window with two eyes in it that follow your cursor around. The eyes also wink when you click your mouse buttons: left and right buttons close the corresponding eye, middle button closes both. Here's what it looks like:



The EYECON window can be shrunk into an icon that looks like this:



doing this kills the process which tracks the cursor. Opening the icon restarts the tracking process.

To start it, load EYECON and call (EYECON.OPEN *left bottom*) to create an EYECON window at (*left bottom*).

INSPIRATION

I hacked this up after hearing about something similar that runs under Suntools. The bitmap for the eyes was borrowed from Spy and slightly modified (thanks, Larry!). Julian Orr suggested that the eyes should wink.

FILEWATCH

Johannes A. G. M. Koomen
(Koomen.wbst@Xerox or Koomen@CS.Rochester)

This document last edited on October 19, 1987.

INTRODUCTION

FILEWATCH is a facility for keeping an eye on open files. It periodically updates a display showing each open file stream, its current file pointer location, the total file size, a percentage bar, and a read/write/both indicator.

DESCRIPTION

Invoking the function FILEWATCH (or selecting the "FileWatch" entry on the BackgroundMenu) starts up the FileWatch process if not already running, or brings up a FileWatch control menu allowing you to forget a currently displayed file (*i.e.*, stop displaying the file), recall a previously forgotten file, close an open file (after mouse confirmation), change some or all FileWatch display properties, or quit the FileWatch process. The Forget, Recall and Close entries on the FileWatch control menu have roll-outs to let you perform the operation on several files at once.

FileWatch can be customized by setting the FileWatch properties (see below) using the function FILEWATCHPROP. Right buttoning any FileWatch window brings up the FileWatch control menu, with the provision that the Forget and Close commands apply to the file displayed in that FileWatch window. Middle buttoning any FileWatch window allows you to move the entire FileWatch display, and left buttoning cause the window to be redisplayed.

DETAILS

(FILEWATCH Command) [Function]

If Command is 'ON and no FileWatch process is already running, starts a process to watch open files. If Status is 'OFF or 'QUIT and there is a FileWatch process running, kills the process. If Command is neither one of the above nor one of the FileWatch commands listed below, starts a process to watch open files if not already running, otherwise brings up the FileWatch control menu. Returns the process if running, otherwise NIL.

FORGET [FileWatch command]

Brings up a menu of files currently being watched. Select the one you no longer want to have watched.

FORGET-MANY [FileWatch command]

Repeatedly performs the FORGET command until no other files are being watched or you make a null selection.

RECALL [FileWatch command]

Brings up a menu of forgotten files. Select the one you want to have watched again.

RECALL-MANY [FileWatch command]

Repeatedly performs the RECALL command until all forgotten files are being watched again or you make a null selection.

CLOSE [FileWatch command]

Brings up a menu of open files. Select the one you want to have closed.

CLOSE-MANY [FileWatch command]

Repeatedly performs the CLOSE command until all open files have been closed or you make a null selection.

MOVE [FileWatch command]

Performs the SET-ANCHOR, SET-POSITION, and SET-JUSTIFICATION commands.

SET-ANCHOR [FileWatch command]

Brings up a menu of four corner names. Select the one on you wish to anchor the FileWatch display. For instance, selecting Top-Right causes FileWatch windows to be stacked downwards witht the top right corner of the first FileWatch window at the FileWatch display position.

SET-POSITION [FileWatch command]

Indicate where the FileWatch display should be positioned by moving the region of the combined FileWatch windows.

SET-JUSTIFICATION [FileWatch command]

Requests confirmation to turn FileWatch window justification on, *i.e.*, make all FileWatch windows the same width as the largest one.

(FILEWATCHPROP PropName [PropValue]) [Function]

If PropValue is given, sets the property value accordingly. Always returns the current (old) value of the property. This is a general facility which you can use for whatever purpose you deem appropriate. However, there are some properties that have a predefined meaning to FileWatch:

ALL-FILES? [FileWatch property]

If NIL, FileWatch displays only user visible open files; otherwise all open files (including, for example, dribble and file cacher files) are displayed. Initially set to NIL. Caveat: setting this property to T will give you access to things that might be dangerous to play with. In particular, closing certain system files on the Dorado may cause your machine to crash, and may leave the local file system in an unhealthy state.

ANCHOR [FileWatch property]

Each open file that is being watched gets its own FileWatch window. Multiple windows are stacked automatically. The total region occupied by this stack is anchored at the corner indicated by this property. The only legal values are TOP-LEFT, TOP-RIGHT, BOTTOM-LEFT, BOTTOM-RIGHT. Initially set to BOTTOM-RIGHT. If the anchor is at one of the bottom corners the stack grows upward, otherwise downward. If the anchor is at one of the left corners the stack is aligned by left edge, otherwise by right edge (see also the JUSTIFIED? property).

FILTERS [FileWatch property]

A list of file patterns, for example '("{CORE}*.*;*"). An open file that matches any of the patterns will not be watched. Initially set to NIL. Note that each pattern is expanded to include the HOST and DIRECTORY equal to that of (DIRECTORYNAME), EXTENSION and VERSION equal to "*", unless already specified. For example, in my case, the filter "*JUNK*" expands to "{Ice}<Koomen>Lisp>*JUNK*.*;*". If you really wanted to filter all junk files, use the filter "{*}*JUNK*".

FONT [FileWatch property]

The font used for the FileWatch displays, specified in a form suitable to give to the function FONTCREATE. Initially set to '(GACHA 8).

INTERVAL [FileWatch property]

The value given to the function BLOCK. This should be either NIL or an integer indicating the number of milliseconds to wait between FileWatch display updates. Initially set to 1000. Note that FileWatch generates several FIXP's for large files every time throught the loop, so setting this to NIL may cause excessive storage allocation and reclamation.

JUSTIFIED? [FileWatch property]

If T all FileWatch windows are aligned along both left and right edges, and are grown or shrunk as needed to accomodate the maximum filename length currently in use. This is aesthetically more pleasing but incurs increased overhead due to frequent reshaping of the windows. Initially set to NIL.

POSITION [FileWatch property]

The location of the anchored corner of the FileWatch display. Initially set to the bottom right corner of the screen: (CONS SCREENWIDTH 0).

SHADE [FileWatch property]

The shade used for the FileWatch thermometers. Initially set to GRAYSHADE.

SORTFN [FileWatch property]

Either NIL or the name of a function taking two filenames as arguments (such as ALPHORDER), which is used to sort the list of open files being watched. Initially set to NIL (*i.e.,* no sorting).

FILLREGION

Originally By: Mike Bird (Inference Corp., Los Angeles, CA)
Jim Wogulis (Wogulis@ICS.UCI.EDU)
Greg Wexler (Wexler.pasa@Xerox)
New Owner: James M. Turner (Turner.Lexington@Xerox.com)

INTRODUCTION

The Fillregion package provides a function which will allow the user to "fill in" arbitrary regions of a bitmap or window with a shade or bitmap (or any valid shade argument to BITBLT).. The regions must be defined by a black or white outline. There are two functions provided to the user: FILL.REGION and AUTO.FILL.

(FILL.REGION *window.or.bm* *interior.pos* *shade*) [Function]

window.or.bm : Must be either a window or bitmap otherwise an error occurs.

interior.pos : Must be a position within window.or.bm that is within the interior of the region to be filled.

shade : Shade can be any valid shade argument that BITBLT will accept.

This will return the window.or.bm with the specified region filled in. The region to be filled is determined by the pixel specified at interior.pos. If the pixel is black, all the connected black regions will be shaded, otherwise, if the pixel is white, all the connected white region will be filled. If the user aborts the function before completion, the orginal window.or.bm will be restored.

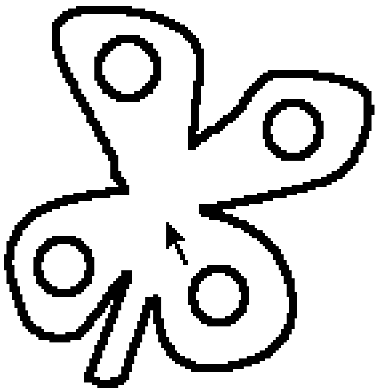
(AUTO.FILL *shade*) [Function]

shade : Shade can be any valid shade argument that BITBLT will accept.

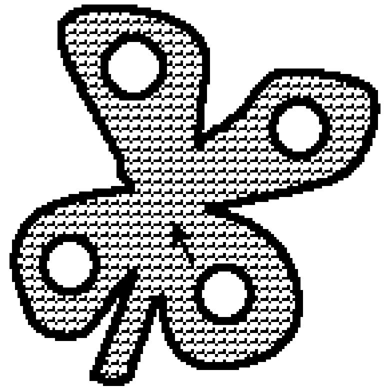
With your mouse pointing inside the appropriate region in a window, this function will fill in the region with the shade specified. This package only works for one bit per pixel bitmaps, color is not supported.

Example:

```
(AUTO.FILL 1234)
```



results in:



Comments and suggetions are welcome.

FINGER

By: Greg Nuyens (mcvax!inria!nuyens@seismo.css.gov)

INTRODUCTION

Finger is a facility for determining and displaying information about other users running Xerox LISP. It displays the user's name, the Etherhostname (or the octal net address when no nameserver is available) and the user's idle time (time since last keystroke or mouseaction). Only other users who have the finger server loaded will be displayed. Users can specify the net radius to query, a list specifying only which users they want displayed, or similarly, only which hosts are to be displayed.

Loading Finger begins the finger server (which responds to queries). To display finger information, the following top-level function is provided:

(FINGER WHO HOST HOPS ICON?) [Function]

WHO is an optional list of usernames specifying which people are to be displayed if a response is received. who defaults to **FINGER.CROWD**, initially **NIL** meaning display all responses.

HOST is an optional list of etherhostnames analogous to *WHO*. Specifying both *WHO* and *HOST* denotes union. *HOST* defaults to **NIL**, denoting all hosts.

HOPS specifies the net radius to query. 0 specifies only nets to which you are directly connected. hops defaults to **FINGER.NET.HOPS**, initially 2.

ICON? specifies whether initial display should be the finger icon

or a display window. *ICON?* defaults to **NIL** meaning display. {typically, in an init file the call would be **(FINGER NIL NIL NIL T)**}.

The display window is updated each time the users bugs the display window with left or middle mouse button, and when most window operations are performed on the display window (shape, repaint, expand from icon, etc.). Right button retains the standard window menu.

<u>Options:</u>

the following are user specifiable variables affecting the operation of Finger.

FINGER.ICON.POSITION a position indicating the original position for the icon. Initially (900,500).

FINGER.DISPLAY.POSITION a position indicating the original position for the display window. Initially (650,325).

FINGER.DISPLAY.HEIGHT height of the display window. Initially 140. The display width is correct for the display format and need not be changed.

FINGER.TIMEOUT milliseconds to wait for the last response packet. Initially 1500.

FINGER.NET.HOPS net radius to be queried.

FINGER.CROWD list of potential users to be displayed (discussed above).

FINGER.INFINITY.MINUTES number of minutes to be considered infinite idle time. Initially 90.

Additional functions of interest to the user are:

(END.FINGER) which kills the finger server process, closes the sockets, closes the windows, etc.

(FINGER.SERVER) will start a finger server process.

FREE MENU CREATOR

By : André Blavier (Rank Xerox France)

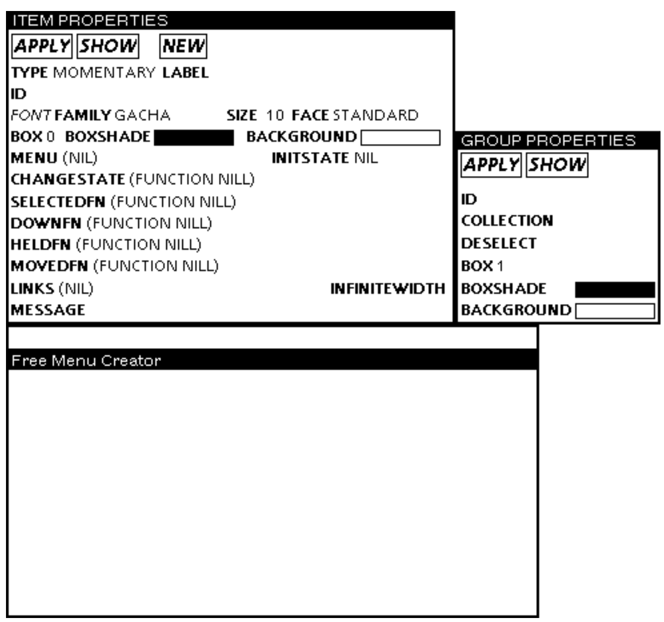
This document last edited on September 29, 1988

The Free Menu Creator application is designed to create interactively Free Menu description lists.

It is a full graphical tool with which you can move, shape, box, group the Free Menu Items and attach properties to them. At any time you can make the application compute the description list and then test the Free Menu that you have just created.

OPENING A FREE MENU CREATOR (FMC) WINDOW

Once you have loaded the application the ’FMCreator’ option is added to the background menu. By selecting it you will create a FMC window of the form :



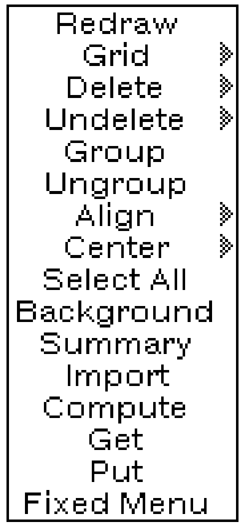
The bottom window is the **main window** : you will add, move ... items to it.

Two menu windows (Free Menus in fact) are attached on top of the main window : the **Item Properties menu** (IP menu) and the **Group Properties menu** (GP menu). They are provided for setting and changing item or group properties.

Between the menus and the main window is the **Prompt window**. Its purpose is mainly to display various information suchas messages, prompts ...

THE RIGHT BUTTON MENU

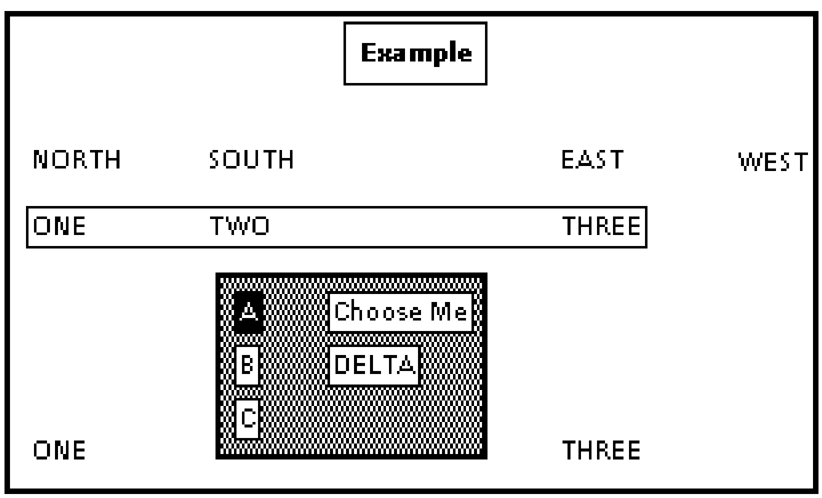
Pressing the right button inside the main window will pop up the following menu :



You can fix this menu on the right edge of the main window by selecting the 'Fixed Menu' option.

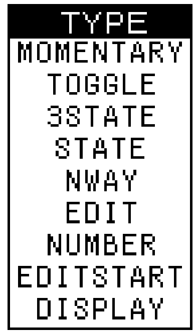
A SAMPLE SESSION

Suppose you want to create the following Free Menu (this example is taken from the FREE MENU Lyric Release Notes) :

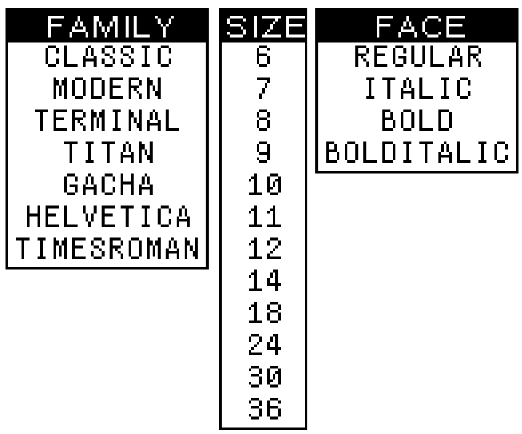


First create the 'Example' item :

- select **TYPE** in the IP menu : the following pop-up menu appears :



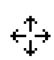
- choose the type of item you want to create, e.g. DISPLAY
- select **LABEL** in the IP menu and type ’Example’
- choose the font by clicking in the **FAMILY**, **SIZE** and **FACE** items, which will cause the following menus to pop up :



- now, the properties you wanted for that new item are set. So click in the **NEW** field and then move the mouse to the place where you want to put the item.

When you release the mouse button the item is fixed in the window.It is surrounded by a gray rectangle which means that it is **selected** :

If you move the mouse inside the selection rectangle the cursor will change to :

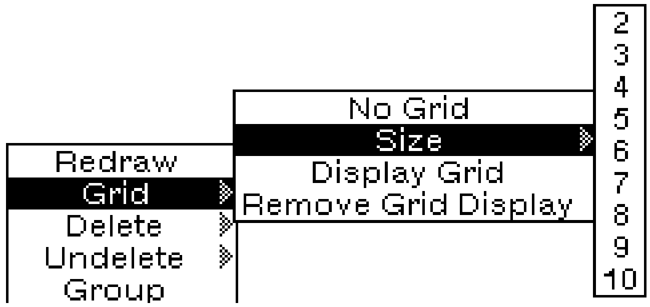


By pressing the left button while the cursor is inside a selection, you can move that selection.

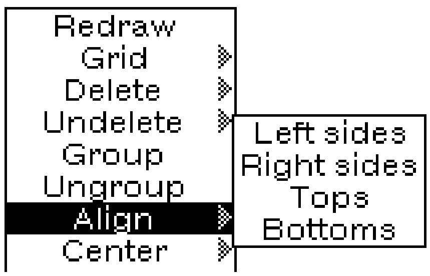
Repeat the same operation for the ’NORTH, ’SOUTH’, ’EAST’, ’WEST’, ’ONE’, ’TWO’ and ’THREE’ items.

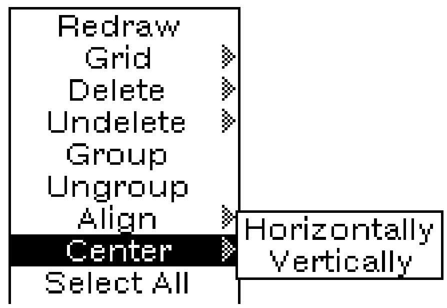
To place the items exactly where you want you can use (and combine) the following facilities :

- use the **GRID** (from the right menu) :

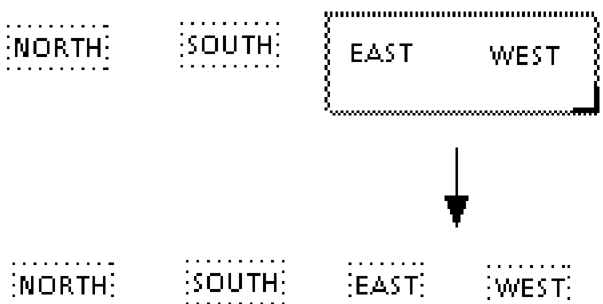


- align and center a **multiple selection** :



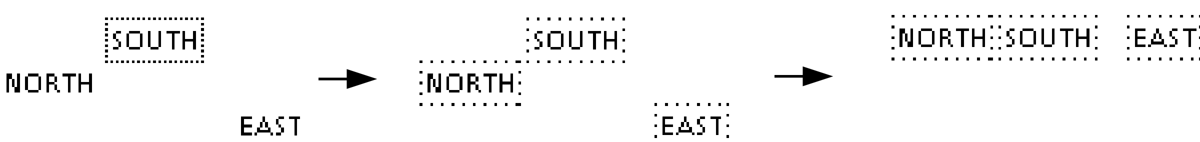


Select multiple items by pressing the left button and the **META** key, while the cursor is above them. You can also extend a multiple selection by pressing Left-Meta outside any item and shaping a ghost region :



Each item of a multiple selection is surrounded by a dotted rectangle. You can move a multiple selection in the same way you move a unique selection.

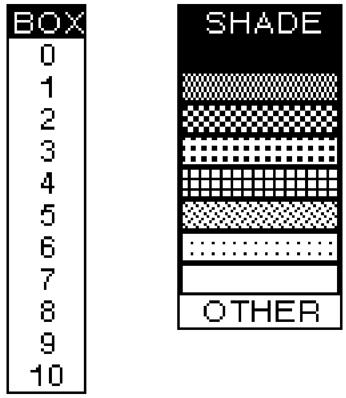
The **Align** and **Center** functions all refer to the first selected item. For example, if you choose 'Align Tops' then all the items except for the first one will be moved so that their top is at the same Y coordinate as that of the first item :



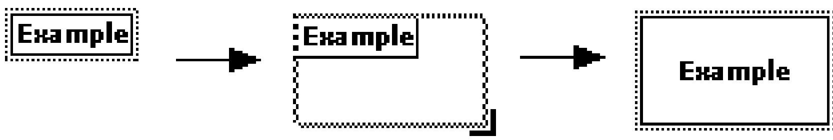
Now, let's box the 'Example' item :

- select the item

- select **SHOW** in the IP menu : this updates the menu according to the item
- set the BOX and BOXSHADE properties :



- select **APPLY** : the item is redisplayed surrounded by a box
- press the middle button inside the item : you can now shape the box (the shaping is controlled so that the space between the item and its box is the same horizontally and vertically) :



Whenever you need to change some items's property (ies) , select the item and then select **SHOW** in the IP menu : this updates the menu according to the item. Then, you can change any property, and update the item according to the IP menu by selecting **APPLY**.

You can also change properties of multiple items : make a multiple selection, set the property (ies) you want to change, then select **APPLY** : the following menu pops up :

Only the selected property will be applied to the items.

When setting a box property on multiple items, a box is created around each item, not around the whole selection. To wrap a set of items by a box you must first **GROUP** the items. That's what we want to do for the 'ONE', 'TWO', 'THREE' items :

- select the items
- choose **GROUP** from the right menu

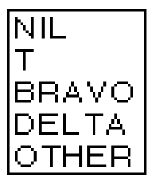
Groups are automatically boxed at creation. You can modify groups in the same way as you modify simple items, using the GP menu. Groups can be moved and shaped like simple items.

Now, create the ’A’, ’B’, ’C’ items, setting their TYPE to NWAY. Group the items, select **SHOW** in the GP menu, set COLLECTION to ’COL1’, set DESELECT to T - then select **APPLY** : you have created the collection.

Create the ’DELTA’ item and then the ’Choose Me’ STATE item. For this last item specify a MENU property, selecting **MENU** in the IP menu. This opens the following SEdit window :

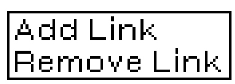
The first element of the list should be an item list, as suitable for standard menus. The second and third elements are optional : FONT should be a list of the form (FAMILY SIZE FACE), TITLE should be a litatom or a string.

Edit the list ((BRAVO DELTA) (MODERN 12 ITALIC)) and close the SEdit window. Now, select **INITSTATE** : this will pop up the following menu :



As you can see the items you just edited are part of this menu. Select DELTA.

One more property is required : the **LINKS** property. Select LINKS : the following menu appears :



Choose ’Add Link’ and click on the DELTA item : the link is created.

At any time you can create the actual Free Menu out of your FMC window by selecting the **COMPUTE** option from the right menu. This creates the description list, which is stored in the **FM-DESCRIPTION** global variable. A call to the FREEMENU function is done automatically with FM-DESCRIPTION as an argument, and the Free Menu window is opened.

REFERENCE GUIDE

General window behavior

The main window, the prompt window and the properties windows behave as a whole. Moving, shaping, shrinking and closing can be directed from anyone of them. Shaping affects only the main window. Closing is protected by a confirmation request when unsaved modifications have occured. The main window is not scrollable.

The right menu, when fixed, can be closed solely.

Hardcopying the main window does not use the standard hardcopy functions. Instead, the contents is 'pretty-hardcopied', but the current version can't print large windows on multiple pages.

The right menu

Redraw : use Redraw to redisplay the contents of the window. This can be useful when items overlap.

Grid : selecting the Grid option displays the grid state in the prompt window.
No Grid : removes the grid.
Size suboptions : specifies a grid size (in pixels). Grid alignment refers to the lower left corner of items.
Display Grid : displays the grid in the window.
Remove Grid Display : removes the grid display, but the grid remains active.

Delete : deletes the selected item(s). Items deleted are saved in a list so they can be undeleted. The number of deleted items is displayed in the prompt window.
Forget save list : deleted items are destroyed and can't be undeleted.

Undelete : undeletes last deleted item.
Last : same effect
All : undeletes all deleted items
List : pops up a menu of all the deleted items. The selected item gets undeleted.

Group : groups a multiple selection. A group can include groups. The components of a group are not individually accesible.

Ungroup : unpacks a group. The ungrouping operation works only at the first level, i.e. included groups are not unpacked.

Align : the Align suboptions all work on a multiple selection (items and/or groups). The alignment operation refers to the first selected item.
Left sides : align left sides of selected items
Right sides : align right sides of selected items
Tops : align tops of selected items
Bottoms : align bottoms of selected items

Center : the Center suboptions all work on a multiple selection (items and/or groups). The centering operation refers to the first selected item.
Horizontally : center items so their center is on the same X coordinate as that of the first selected item
Vertically : center items so their center is on the same Y coordinate as that of the first selected item

Select All : selects all the window contents

Background : pops up a shade menu for setting the window background shade

Summary : creates a TEdit window listing a summary of the window contents. Only interesting properties are described, depending on the item type. Groups contents are indented hierarchically.

Import : allows items importation from a Free Menu. This function works but is currently bugged : importing items with a MENU property is not supported ; when importing groups only their contents are imported.

Compute : generates a description list of the items suitable for the FREEMENU function. The list is stored in the FMC-DESCRIPTION global variable. Opens a Free Menu built out from FMC.

Get : loads the contents of a FMC window, previously stored on disk by a Put operation. The loaded items (groups) are added to the current contents of the window.

Put : saves the contents of a FMC window on disk.

Fixed Menu : when the right menu is poped up by the right mouse button, attaches the menu to the main window. Once the menu is fixed, has no effect.

The Item Properties menu

APPLY : if the selection is unique sets the selected item properties to the properties described in the menu. If the selection is multiple, pops up a menu of properties and sets the selected property of all the selected items according to the menu.

SHOW : updates the menu according to the selected item, thus allowing editing its properties.

NEW : creates a new item which properties are described in the menu.

TYPE : lets the user specify the TYPE property from a pop-up menu.

LABEL : lets the user edit the LABEL property. If no label is edited the item will be displayed in FMC with the pseudo-label '*NOLABEL*'.
Right-buttoning in this field will start label edition, clearing the field first.
Middle-buttoning puts the cursor in the GETREGION state, and sets the label to the bitmap specified by the user.

ID : lets the user edit the ID property. The edited string is always MKATOMed in the description list.

FONT : FAMILY, SIZE and FACE allow font descriptions from pop-up menus.

BOX : BOX=0 means no box.

BOXSHADE, ***BACKGROUND*** : pop up a shade menu, including an 'OTHER' option for shade editing.

MENU : opens a SEdit window where the user specifies the MENUITEMS, MENUFONT and MENUTITLE properties. The list must be of the form (MENUITEMS [MENUFONT] [MENUTITLE]).

INITSTATE : pops up a menu of possible INITSTATE values, depending on the item TYPE and possible MENUITEMS.

CHANGESTATE, ***SELECTEDFN***, ***DOWNFN***, ***HELDFN***, ***MOVEDFN*** : open a SEdit window for the corresponding property. The edited list can be of the form (FUNCTION function-name) or a LAMBDA-expression.

LINKS : pops up a menu with 2 items : 'Add Link' and 'Remove Link'. 'Add Link' prompts the user to click on the item that is to get linked (this item must have an ID). 'Remove Link' removes the link.

INFINITEWIDTH : this toggle item is suitable for EDIT or NUMBER items only.

MESSAGE : starts editing the MESSAGE property (right buttoning clears the field first).

The Group Properties menu

APPLY, ***SHOW***, ***ID***, ***BOX***, ***BOXSHADE***, ***BACKGROUND*** behave in the same way as in the IP window.

COLLECTION : starts editing a COLLECTION name. Specifying a COLLECTION property is suitable only for groups which items are NWAY.

DESELECT : a toggle item whose value is relevant only for COLLECTIONs.

About item shaping

The effects of middle buttoning inside an item is different wether the item is boxed or not.
If the item is boxed, shaping will be constrained as described in the sample session, i.e. BOXSPACE must be the same horizontally and vertically.
If the item is not boxed, shaping is constrained so that only the item's width can be changed, in fact modifying the MAXWIDTH property.

Copy functions

The effects of pressing the COPY key and left buttoning when a FMC window has the TTY depend on what is to be copied :

- when clicking in a SKETCH window, the selected objects are made a graphic LABEL

- when clicking in a FMC window, the selected items are copied in the target window (the source and the target can be the same).

FONTSAMPLER

By: Nick Briggs (Briggs.pa@Xerox.com)

This document last edited on April 27, 1987

INTRODUCTION

The FONTSAMPLER package provides a function which will easily generate sample sheets of your favourite fonts on your favourite printer.

FUNCTIONS

(FontSample *Fonts CharacterSets Printer StreamType*) [Function]

Creates a sample sheet for the font described by the font descriptor(s) *Fonts* on the image stream generated by passing *Printer* and *StreamType* to OPENIMAGESTREAM. *Fonts* may be either a single font descriptor (the result of a call to FONTCREATE) or a list of font descriptors. *CharacterSets* may be either a single number, or a list of numbers representing character sets. The default value of *CharacterSets* (if NIL is passed) is 0 (the Roman Alphabet and Punctuation). FontSample will create a page for each font and character set combination specified, consisting of all the characters of the font/character set combination, arranged on a 16 by 16 grid, and labelled in the default 12pt font for the stream.

For example, to create a sample sheet for the Modern 12pt regular font, character sets 0 and 239 (general and technical symbols 2) on the Interpress printer Fountain one might do:

```
(FontSample (FONTCREATE 'MODERN 12 'MRR 0 'INTERPRESS) '(0 239) '{LPT}Fountain:)
```

(FontSampleFaked *FontAsList Printer StreamType*) [Function]

There are very often fonts on printers that Interlisp-D cannot speak about, because it is lacking a font width file. FontSampleFaked can produce sample sheets for many of these fonts. *FontAsList* should be a font expressed in the raw form that FONTCREATE takes, *Printer* and *StreamType* are as for FontSample, however passing DISPLAY for *StreamType* will not produce meaningfull results.

To produce a sample sheet for the Vintage 10pt bold font on the Interpress printer Fountain one might incant:

```
(FontSampleFaked '(Vintage-Printwheel 10 BRR) '{LPT}Fountain:)
```

NOTES

Be careful if you create sample sheets for many fonts at once – especially the larger type sizes (24pt and up) – you may cause an Interpress printer to hang and require rebooting to recover.

Subject: Interlisp to Commonlisp conversion package
From: darrelj%sm.unisys:COM
To: info-1100%sumex-aim.stanford

FORMACRO and FORMACRO.DFASL -- Still another portable iteration macro for commonlisp. Its main claims are almost 100% compatibility with the semantics of the Interlisp-Clisp FOR (especially when used the the XFORMS which fix a few incompatibilities); and user extensibility (unfortunately not compatible with IL:I.S.OPR). Embedded keywords (e.g. IN, COLLECT) may be in any package.

FTPSERVER-MULTI-CONNECTIONS

By: Matt Heffron (BEC.HEFFRON@ECLA.USC.EDU)

Requires: FTPSERVER, FTPSERVERPATCH and DPUPFTPPATCH

INTRODUCTION

This package (actually a complimentary pair of files) extends the capabilities of the Lisp Library FTPSERVER package to support multiple simultaneous connections between Xerox 11xx series AI workstations.

INSTALLATION

To install this package, load the FTPSERVERPATCH.LCOM file on the 11xx machine(s) that are to be servers (this will load FTPSERVER if it is not already loaded). Then load the DPUPFTPPATCH.LCOM file on any of the 11xx machines that are to be clients of these servers. You must set the value of IL:*FTP.NEGOTIATED.CONNECTION.HOSTS* on each of the client machines to specify the server machines that support the FTPSERVERPATCH system of multiple simultaneous connections (below).

VARIABLES

IL:*FTP.NEGOTIATED.CONNECTION.HOSTS* [Global Variable]

This variable must be set to specify the server machines that support the FTPSERVERPATCH system of multiple simultaneous connections. Its value is a list of PUP host numbers. (Specifically, it is a list of the values of `(CAR (BESTPUPADDRESS` <SERVER–HOST–NAME>`))` for each of the server machines.)

HOW IT WORKS

This package modifies the DPUPFTP code of the client machines, so that when it is trying to open an FTP connection BSP stream, it first checks to see if the server host is one of the IL:*FTP.NEGOTIATED.CONNECTION.HOSTS*, and if so, it sends a message to the modified FTPSERVER on that system (using PUP type \PT.NEGOTIATED.CONNECTION (= 128) on PUP socket \PUPSOCKET.NEGOTIATED.CONNECTION (=63)). The server machine creates a socket for this connection and starts a standard FTPSERVER listener process on this socket, and returns the socket number to the client. (The process is modified so it will go away when the connection is closed instead of lingering forever.) The client uses the returned socket number for the connection instead of \PUPSOCKET.FTP. If the server is NOT on IL:*FTP.NEGOTIATED.CONNECTION.HOSTS*, or fails to respond within 10 seconds with the new socket number, then \PUPSOCKET.FTP is used. When the negotiated connection server is started on the server machine (with the incantation `(FTPSERVER)` which is the original FTPSERVER start up), it also will start up a *permanent* FTPSERVER listener on \PUPSOCKET.FTP so regular connection requests can be handled.

ACKNOWLEDGEMENTS

Thanks to Tom Lipkis of Savoir for suggesting this sort of scheme.

GITFNS

By Ron Kaplan

This document was last edited in May 2022.

GITFNS provides a Medley-oriented interface for comparing the files in two different branches of a git repository. This makes it easier to understand what functions or other definitions have changed in a Lisp source file, or what text has changed in a Tedit file. This may be particularly helpful in evaluating the changes in a pull request.

Separately, GITFNS also provides tools and conventions for bridging between git's file-oriented style of development and version control and Medley's residential development style with its own version control conventions. GITFNS allows for intelligent comparisons between Lisp source files,Tedit files, and text files in a local git clone and a local Medley-style working directory, and for migrating files to and from the git clone and the working directory.

Git projects: Connecting git clones to GITFNS capabilities

The GITFNS capabilities operate on pre-existing clones of remote git repositories that have been installed at the end of some path on the local disk. The path to a clone can be used to create a "git project" for that clone:

```
(GIT-MAKE-PROJECT PROJECTNAME PROJECTPATH WORKINGPATH EXCLUSIONS
                  DEFAULTSUBDIRS)     [function]
```

where

PROJECTNAME is the name of the project (e.g. MEDLEY, NOTECARDS, LOOPS...)

PROJECTPATH is the local path to the clone (e.g. {dsk}<users>...>git-medley)

WORKINGPATH is optionally the local path to a corresponding Medley-residential working directory (e.g. {dsk}<users>...>working-medley>)

When the project has a WORKINGPATH:

EXCLUSIONS is a list of files and directories to be excluded from comparisons (beyond what its .GITIGNORE specifies)

DEFAULTSUBDIRS is a list of subdirectories to be use in working-path comparisons when directories are not otherwise specified.

For convenience, if PROJECTPATH is NIL or T (and not a path), then a squence of probes based on PROJECTNAME attempts to find a clone directory (with a .git subdirectory):

```
(UNIX-GETENV PROJECTNAME)

(UNIX-GETENV (CONCAT PROJECTNAME 'DIR)

(CONCAT MEDLEYDIR "../git-" PROJECTNAME)
    (a sister of MEDLEYDIR named git-PROJECTNAME, e.g. git-notecards)
```

Thus:

If MEDLEYDIR is defined,
(GIT-MAKE-PROJECT 'MEDLEY) will make the MEDLEY project
If NOTECARDS is defined
(GIT-MAKE-PROJECT 'NOTECARDS) will make the NOTECARDS project
If NOTECARDS is not defined but the clone >git-notecards> is a sister of MEDLEYDIR, then the NOTECARDS project will still be created.

If a clone is discovered and a project is created, the value of GIT-MAKE-PROJECT is PROJECTNAME. Otherwise, NIL will be returned if PROJECTPATH is T (= no-error), and PROJECTPATH=NIL will result in an error.

When GITFNS is loaded, GIT-MAKE-PROJECT is called for MEDLEY, NOTECARDS, and LOOPS, with PROJECTPATH=T. Thus, those projects will be created automatically, if MEDLEYDIR is defined and the relevant directories exist in their expected relative positions.

When they are created, GIT-PROJECTS are registered by name on the a-list GIT-PROJECTS, and they can otherwise be referenced by their names.

The variable GIT-DEFAULT-PROJECT, initially MEDLEY, contains the project name used by the commands below when the optional projectname argument is not provided.

GIT-MAKE-PROJECT also creates a pseudohost {Gprojectname} whose path prefix is the prefix for the project's clone. If WORKINGPATH is provided, then a second pseudohost {Wprojectname} points to the working files for the project.

GITFNS also defines two directory-connecting commands for conveniently connecting to the git and working pseudohosts of a project:

cdg (projectname) (subdir) [command]

cdw (projectname) (subdir) [command

For example, cdg notecards library connects to {GNOTECARDS}/library/.

Comparing directories and files in different git branches

In its simplest application, GITFNS is just an off-to-the-side add-on to whatever work practices the user has developed with respect to a locally installed git project. Its only advantage is to allow for more interpretable git-branch comparisons, especially for pull-request approval. These comparisons are provided by the prc ("pull request compare") Medley executive command:

prc (branch) (DRAFT) (projectname) [command]

This compares the files in branch against the files in the main branch of the project (origin/master or origin/main). Thus, suppose that a pull request has been issued on github for a particular branch, say branch rmk15 of the default project. Then

prc rmk15

brings up a lispusers/COMPAREDIRECTORIES browser for the files that currently differ between origin/rmk15 and origin/master. If the selected files are Lisp source files, the Compare item on the file browser menu will show the differences in a lispusers/COMPARESOURCES browser. The differences for other file types will be shown in a lispusers/COMPARETEXT browser.

If branch is not specified and the shell command gh is available, then a menu of open pull-request branches will be provided. If gh is not available, the menu will offer all known branches. If the optional DRAFT is provided, then the menu will include draft PR's as well as open ones.

If one PR, say rmk15, contains all the commits of another (rmk14), then the menu will indicate this by

rmk15 > rmk14

Note that the prc comparison is read-only: any comments, approvals, or merges of the branch must be specified using the normal Medley-external git interfaces and commands.

prc is the special case of the more general bbc command ("branch-branch compare) for comparing the files in any two branches:

bbc branch1 branch2 (project) [command]

This compares the files in branch1 and branch2, for example

bbc rmk15 lmm12 (local)

This will compare the files in origin/rmk15 and origin/lmm12 in the GIT-DEFAULT project. branch1 defaults to the origin files of the currently checked out branch, the second defaults to origin/master. If local is non-NIL, then a branch that has neither local/ or origin/ prepended will default to local (e.g. local/rmk15) instead of origin/. Local refers to the files that are currently in the clone directory, which may not be the same as the origin files, depending on the push/pull status.

Either of the branches can be specified with an atom LOCAL, REMOTE, or ORIGIN, in which case bbc will offer menus listing the currently existing branches of that type.

NOTE: Branch comparison makes use of a git command that has a limit (diff.renameLimit) on the number of files that it can successfully compare. A message will be printed if that limit is exceeded, asking whether a larger value for that limit should be applied globally.

The command cob ("check out branch") checks out a specified branch:

cob branch (nexttitlestring) (project) [command]

This checks out branch of project and then executes git pull. The branch parameter may also be a local branch, T (= the current working branch), or NEW/NEXT (= the next working branch). The current working branch is the branch named <initials>nnn, e.g. rmk15. The initials are the value of INITIALS as used for SEDIT time stamps, and nnn is the largest of the integers of all of the branches beginning with those initials.

If branch is NEW or NEXT, then a new initialed branch is created and becomes the user's current branch. Its number is one greater than the largest number of previous initialed branches. If nexttitlestring is provided, then that string will be appended to the name of the branch, after the initials and next number, and two hyphens. Spaces in nexttitlestring will also be replaced by hyphens, according to git conventions.

If branch is not provided, a menu of locally available branches pops up.

The currently checked out branch is obtained by the b? command:

b? (project) [command]

Correlating git source control with separate Medley development

It is generally unsafe to do Medley development by operating with files in a local clone repository. Medley provides a residential development environment that integrates tightly with the local file system. It is important to have consistent access to the source files of the currently running system, especially for files whose contents have been only partially loaded. A git pull or a branch switch that introduces new versions of some files or removes old files altogether can lead to unpredictable disconnects that are hard to recover from. This is true also because development can go on in the same Medley memory image for days if not weeks, so it is important to have explicit control of any file version changes.

GITFNS mitigates the danger by conventions that separate the files in the git clone from the files in the working Medley development directory. The location of the Medley development source tree for a project is given by the WORKINGPATH argument to GIT-MAKE-PROJECT. If WORKINGPATH is T or NIL and there exists a directory >working-projectname> as a sister to the clone, then that is taken to be the WORKINGPATH and thus the prefix for a pseudohost {Wprojectname}.

When Medley development is carried out in the WORKINGPATH, the variable MEDLEYDIR should point initially to the working directory, and the directory search paths (DIRECTORIES, LISPUSERSDIRECTORIES, FONTDIRECTORIES, etc.) all have MEDLEYDIR (or {WMEDLEY}) as a prefix. In that case, the clone for the project, if PROJECTPATH doesn't specify it explicitly, should be located at the >git-medley> sister directory of MEDLEYDIR.

Any back and forth transfer of information between the git clone and Medley development must be done by explicit synchronization actions. Crucially, Medley-updated files do not appear in the clone directories and new clone files do not move to the Medley directories without user intervention.

The files in Medley working tree and the git clone of a project can be compared with the gwc ("git-working-compare") command:

gwc subdirectories (project) [command]

This produces a browser for all the files in the corresponding WORKINGPATH subdirectories that differ from the files in the currently checked out branch of the git clone. If subdirectories is omitted, it defaults to the DEFAULTSUBDIRS of the project. If it is ALL, then files in all subdirectories that are not found in the project's EXCLUSIONS are compared.

In addition to the commands for comparing and viewing files, the menu for this browser also has commands for copying files from the git clone {Gprojectname} to {Wprojectname} and deleting files from {Wprojectname}.

If the master/main branch is the current branch then the menu has no commands to change the clone directory. The browser will show those files that have been updated from a recent merge, and they can individually be copied from the git branch to realign the two source trees with incremented Medley version numbers. If the comparison is with a different branch, say the user's current staging branch, copying files from the working Medley to the git clone or deleting git files will set git up for future commits.

Note that the menu item for deleting Medley files will cause all version to be removed, not just the latest one, to avoid the possibility that an earlier one is revealed. Deletion for Medley files is also accomplished by renaming to a {Wprojectname}<deletion> subdirectory so that they can be recovered if a deletion is in error. Files in the git-clone are removed from the file system immediately, since git provides its own recovery mechanism for those files.

GITFNS does not (yet?) include functions for commits, pushes, or merges for updating the remote repository. Those have to be done outside of Medley through the usual github interfaces, as guided by the information provided by the comparisons.

GRAPHCALLS

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

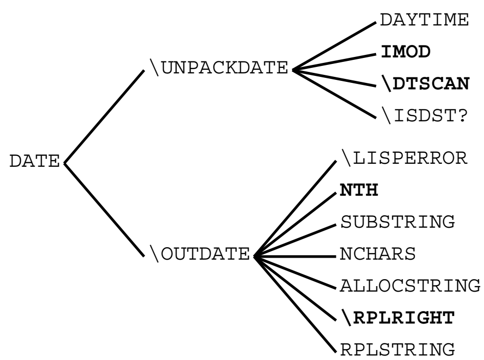
Uses: GRAPHER, MSANALYZE (WHERE-IS & HELPSYS optional)

GRAPHCALLS is an extended graphical interface to the Envos Lisp CALLS function. It is to CALLS what BROWSER is to SHOW PATHS in MASTERSCOPE. It allows fast graphing of the calling hierarchy of both interpreted and compiled code, whether or not the source is available (see the CALLS function in the MASTERSCOPE section of the *Lisp Library Modules* manual), allowing examination of both user and system functions. The sources of the functions do not have to be analyzed by MASTERSCOPE first.

Additionally, buttoning a function on the graph brings up a menu of operations that can be done with the function, such as editing, inspecting, further graphing etc.

(GRAPHCALLS *FUNCTION* &REST *OPTIONS*) [Function]

Graphs the calling hierarchy of *FUNCTION*. Terminal nodes on the graph (those which call no other functions or are undefined) are printed in a bold version of the graph's font indicating that they cannot be graphed further:



The remainder of the arguments, in keyword format, make up *OPTIONS* eg.

```
(GRAPHCALLS 'DATE :FONT '(GACHA 10) :DEPTH 4 :FILTER 'FGETD)
```

Options include:

:STREAM An image stream to display the graph on. The options list is saved on the stream.

:FILTER A predicate to apply to the functions when building the graph to test their eligibility to appear on the graph. The filter can be any defined function; the default is not to filter. Interesting filters include:

WHEREIS Limits the tree to only functions the user as has loaded and prunes out system functions and SYSLOADed files. Quite useful.

FGETD Limits the tree to only functions that are actually defined. Thus if you are perusing the tree for BITBLT and do not have and are not interested in the color code, FGETD will remove all of the undefined color bitmap functions.

EXPRP Limits the tree to interpreted functions. Useful for graphing functions in the development stage.

CCODEP Limits the tree to compiled functions.

NO Keeps low level functions starting with \ (i.e. \OUTDATE) off of the graph. Useful for getting an overview of system functions and when advising system functions (as \'ed functions should probably not be advised).

:DEPTH The calling hierarchy is graphed to *depth* levels (defaults to 2).

:FORMAT Passed to LAYOUTGRAPH and can be any format specification (LATTICE, VERTICAL, REVERSE etc.); defaults to (HORIZONTAL COMPACT REVERSE/DAUGHTERS). In the forest format multiple instances of a function appear on the graph after every calling function and a boxed node indicates the function appears elsewhere on the graph, possibly graphed further. In the lattice format each function gets placed on the graph only once (particularly useful for dynamic graphing, described below), and boxed nodes indicate recursive functions calls.

:SEARCHFN A function to use to generate the children of a given node. It should return a list whose first item is a list of the children, the other items in the list are ignore. Using this feature, it is possible to graph things other than functions. To graph what files load other files, supply a search function of `(LAMBDA (FILE) (LIST (FILECOMSLST FILE 'FILES)))` and a file name for the function argument.

:ADVISE Advises the functions after they are graphed (see *Dynamic Graphing* below); recognized values are one or both of the following:

INVERT Visually tracks a running program .

COUNT Counts function calls in a running program.

:DELAY The delay to use in advised graphs; defaults to 500 milliseconds.

:NAMEFN A function to use to generate the node labels on the graph.

:FONT The font to use to display the graph; defaults to (GACHA 8).

:SHAPE A boolean that indicates if the window should be shaped to fit the graph; defaults to NIL.

:PRIN2FLG A boolean that indicates to use PRIN2 when printing node labels, defaults to NIL.

:SUBFNDEFFLG A boolean that enables graphing of compiler generated functions; defaults to T.

:TOPJUSTIFYFLG Passed to SHOWGRAPH; defaults to NIL.

:ALLOWEDITFLG Passed to SHOWGRAPH; defaults to NIL.

GRAPH MENUS

The menu that pops up when you left button a function on the graph contains the following items:

?= Print the arguments to the function, if available.

HELP Calls HELPSYS on the function.

FNTYP Print the function's FNTYP.

WHERE Do a WHEREIS (with FILES = T) on the function.

EDIT Calls the editor on the function if available for editing.

TYPEIN BKSYSBUFs the name of the function into the typein buffer.

BREAK Applies BREAK to the function. Its subitems are:

- **BREAKIN** Breaks the function only in the context of a particular calling function. In lattice format, if the function has more than one function calling it on the graph, the user is prompted to indicate the caller in which to break the function.
- **UNBREAKIN** Undoes BREAKIN.
- **UNBREAK** Applies UNBREAK to the function.
- **TRACE** Applies TRACE to the function.
- **TRACEIN** Traces the function only when called from inside a particular function, like BREAKIN above. Use UNBREAKIN to remove the trace, or else UNBREAK on the window menu.

CCODE Calls INSPECTCODE on the function if it is compiled code.

GRAPH Calls GRAPHCALLS to make a new graph starting with function, inherits the original graph's options.

FRAME Inspect the local, free and global variables of the function. These are the last three lists of the CALLS function placed into INSPECT windows. Its subitems are:

- **>FRAME** Like FRAME but for all of the functions on the sub-tree starting at the selected node and only for FREEVARS and GLOBALVARS.
- **<FRAME** Like >FRAME but for all of the functions above the function in the graph, i.e. the FREEVARS and GLOBALVARS in the function's scope.

Buttoning the graph outside a node give you a menu with these options:

UNBREAK Does an (UNBREAK), unbreaking all broken functions.

RESET Resets the counters for the COUNT option and redisplays the graph.

DYNAMIC GRAPHING

When the ADVISE option is specified with the value(s) of INVERT and/or COUNT, GRAPHCALLS will advise all of the functions on the graph (in the context of their parent) to invert their corresponding node on the graph (as well as delay some to allow it to be seen) and/or follow each function name by a count of the number of times it has been executed. In invert mode, a node remains inverted as long as control is inside its corresponding function and it returns to normal when the function is exited. The lattice format is best when using the invert feature. Closing the graph window UNADVISEs the functions on the graph.

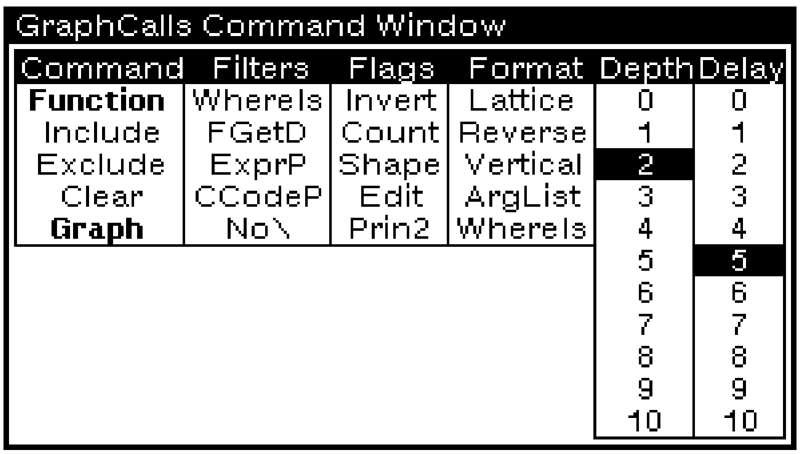
An example of this is `(GRAPHCALLS 'DATE :ADVISE 'INVERT)` and then evaluate `(DATE)`.

GRAPHCALLS will not graph or advise any function in the system list UNSAFE.TO.MODIFY.FNS when the advise option is used. Functions which are unsafe to advise should be added to this list.

CAVEAT PROGRAMMER! This feature must be used with caution. As a rule, one should not do this to system functions, but only one's own, use WHEREIS as a filter for this. Advising system code indiscriminately will probably crash the machine unrecoverablely.

You can, at some risk, interactively break and edit functions on the graph while the code is executing. Also, creating subgraphs of advised graphs will show the generated advice functions not the original functions called, as will creating new graphs of functions in advised graphs. You can create advised graphs of functions already graphed normally on the screen.

COMMAND WINDOW



(GRAPHCALLSW *[REBUILD?]*) [Function]

Puts up a command window with menus that will interactively set up calls to GRAPHCALLS. The menus let you set the Invert, Count and Edit flags, select from common filters and formats and set the depth of the graph. You can also change the amount of delay used in the advised functions when doing dynamic graphing. If you specify an advised graph (Invert or Count) and do not specify a WHEREIS filter, you will be asked to confirm with the mouse for your own protection.

More than one item on the filter and flags menus can be selected at a time. Buttoning a selected item on these menus unselects it. The command menu contains the following:

Function Prompts for the name of a function to graph when the **Graph** item is selected.

Include Adds files or functions to the list of items to allow on the graph, see the Include/Exclude algorithm below.

Exclude Adds files of functions to the list of items disallowed on the graph, see the Include/Exclude algorithm below.

Clear Clears all of the settings on the command window to their defaults. Also clears the Include/Exclude lists.

Graph Graphs the function by calling GRAPHCALLS with the selected options.

Include and Exclude allow fine tuning of the filter function. If the function passes the filter, then the following are tried until one determines whether or not the function will be on the graph:

If a set of functions has been explicitly excluded, and the function is a member of this set, it will NOT appear on the graph.

If a set of functions has been explicitly included, and the function is a member of this set, it WILL appear on the graph.

If a set of files has been explicitly excluded, and the function is in one of those files, it will NOT appear on the graph.

If a set of files has been explicitly included, and the function is not in one of those files, it will NOT appear on the graph.

The function WILL appear on the graph.

The format menu contains two items that are not passed on to GRAPHER but are used to select alternate NAMEFN options:

ArgList Supplies a NAMEFN that will print the function and its arguments (using SMARTARGLIST) as the node label.

WhereIs Supplies a NAMEFN that will print the function followed by the file(s) found by doing WHEREIS (with FILES = T) if any .

When the command window is open, middle buttoning a node on a GRAPHCALLS graph will bring up a menu of commands relating to command window and graphs. The menu contains:

EXCLUDE Adds the function to the exclude functions list of the command window. This is the only way to exclude system functions which get added to the SYSTEM file exclusion list.

The command window can also be obtained via the background menu. Subsequent calls to GRAPHCALLSW (either directly or via the background menu) will reuse the old command window if there is one. If this window is damaged, and redisplay does not help, then setting *REBUILD?* to T will build a new command window from scratch.

NOTES

- Function call graphs are constructed using breadth first search but GRAPHER lays out graphs depth first so functions may be expanded in different places on the graph than expected.
- GRAPHCALLS sysloads GRAPHER and MSANALYZE if they are not already loaded.
- In dynamic graphs, variables caused by advising show up in the frame inspections.
- The global variable GRAPHCALLS.DEFAULT.OPTIONS contains all of the defaults for GRAPHCALLS keywords, in property list format.

GraphGroup

By: Nick Briggs (Briggs.pa@Xerox.com)

For internal use only.

The GraphGroup package contains functions for generating a graph showing the structure of a Grapevine distribution list.

DESCRIPTION

There is a single user-callable function:

(GraphGroup *GroupName InfoStream LayoutOptions ExpandNSGroups*) [Function]

GroupName is the group to be graphed.

Because it is rather slow tracking down the whole structure of a group, if *InfoStream* is non-nil GraphGroup will print on *InfoStream* a trace consisting of: a "[" followed by the number of entries in the group each time it starts a (sub-) group, a "." or "?" each time it identifies an individual , and a "]" at the end of a group. T is a good choice for this parameter.

LayoutOptions is a list of options, in property list format, to be passed to the Grapher function LAYOUTSEXPR. Properties recognized are: FORMAT, BOXING, FONT, MOTHERD, PERSONALD, and FAMILYD. See the Grapher documentation for a description of how to use these options. In most cases you will get a satisfactory graph by leaving this parameter NIL. Should you be planning to hardcopy the resulting graph on an NS printer it is reccomended that if you specify a font you use an NS font (such as Classic or Modern) so that the layout spacing is done correctly.

If *ExpandNSGroups* is T, then GraphGroup will attempt to trace into the NSworld when it finds Grapevine RNames that it recognizes as having pseudo-registries that are really NS domain/organizations.

GREP

By: Larry Masinter (Masinter.pa@Xerox.com)

Requires: BSEARCH

INTRODUCTION

like FGREP of Unix: searches for strings in files.

(GREP *STRS FILES*) [Function]

STRS is a string or a list of strings. FILES is a file or a list of files. Searches for the given string(s) in the given file(s), showing each line.

(PHONE *name*) [Function]

Calls (GREP name PHONELISTFILES). PHONELISTFILES is initialized to NIL. (The PARC init file resets it to point to the PARC phone list.)

For example,

```
(GREP (QUOTE ED) (QUOTE {INDIGO}<REGISTRAR>PARCPHONELIST.TXT))
```

will print:

(from {INDIGO}<REGISTRAR>PARCPHONELIST.TXT;3)

4183 <Endicott>, Fr**ed** 35-1354
4435 <Fiala>, **Ed** 35-2166
4598 <RKenn**ed**y>, Ray 34-78
4839 <McCreight>, **Ed** 35-2146
5759 P**ed**ersen, Jan 32-202
4818 Satterthwaite, **Ed** * 35-2174
MES Solcz, **Ed**ward J. 8* 348-1214
ATA Wahlenmeier, Fr**ed** 887-4018

GRID-ICONS

By: sML (Lanning.pa@Xerox.com)

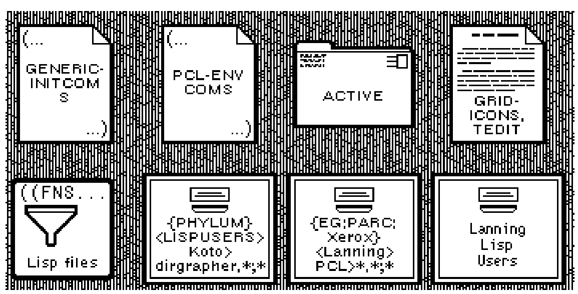
Last edited: September 14, 1987

INTRODUCTION

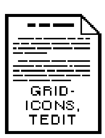
Grid-Icons provides the Lisp user with a set of default window icons that resemble those found in the Viewpoint system. There is an option that the user can set to force these icons to be positioned on a grid, instead of the unrestricted positioning allowed by Lisp.

USING GRID-ICONS

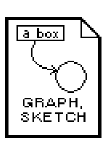
All that is required is loading the file GRID-ICONS. When the file is loaded, it redefines a number of standard window icons in the system.



REDEFINED ICONS



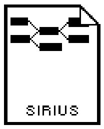
[TEdit icon]



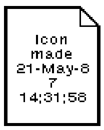
[Sketch icon]

Note that GRID-ICONS not only redefines TEdit and Sketch icons, it also changes the way that the icon title is computed: the host and directory information is removed, so that only the name and extension remain.

[SEdit icon]

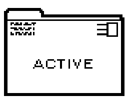


[Loops Browser icon]



[Default icon]

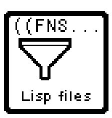
In Lyric, it is possible to redefine the standard icon used by the system. GRID-ICONS uses this, and redefines the standard system icon in LYRIC.



[Lafite mail folder icon]

NEW ICONS

GRID-ICONS defines a handy icon for accessing the list of files that have been loaded into you system.



[Lisp files icon]

Buttoning on this icon will pop up a menu of all loaded files (as determined by the value of the variable FILELST). Selecting a file from this menu will open up an editor on the COMS of that file. There is an additional item on the menu, "* New file *", that can be used create a new file, and then edit its COMS. This icon window is stored in the variable LOADED-FILES-ICON-WINDOW.

USER FUNCTIONS

The user can declare that any given window stick to grid positions.

(GRID-WINDOW window) [Function]

Causes window to pay attention to ENFORCE.ICON.GRID; if the value of ENFORCE.ICON.GRID is true, the window will make sure that it is centered on a grid location. By default the spy button and icons produced by the ICONW and TITLEICONW functions pay attention to ENFORCE.ICON.GRID.

VARIABLES THAT CONTROL GRID-ICONS

There are a few variables that control how GRID-ICONS works.

ENFORCE.ICON.GRID [Variable]

If ENFORCE.ICON.GRID is true, window icons (and any window declared "gridded" by the GRID-WINDOW function) will be restricted to be positioned on a grid. The default value of ENFORCE.ICON.GRID is NIL.

IGNORE.ICON.GRID [Window property]

The IGNORE.ICON.GRID window property provides a way to control icon gridding on an icon-by-icon basis. If the IGNORE.ICON.GRID window property of an icon is true, the icon will not be restricted to grid positions. This window property is checked only if ENFORCE.ICON.GRID is true.

ENFORCE.ICON.REGIONS [Variable]

You can enforce icon gridding in individual regions of the screen by using the variable ENFORCE.ICON.REGIONS. If the value of ENFORCE.ICON.GRID is true, and the icon does not have a IGNORE.ICON.GRID window property, the proposed new position for the icon is tested against the value of ENFORCE.ICON.REGIONS. If ENFORCE.ICON.REGION is NIL, gridding is enforced as described above. Otherwise, ENFORCE.ICON.REGION should be a list of regions; gridding will be enforced only if the proposed position is within one of these regions. The default value of ENFORCE.ICON.REGIONS is NIL. [Thanks/blame to Ramana Rao for this.]

ICON.SIZE [Variable]

ICON.SIZE specifies the maximum size of the icons, for use in computing the grid positions of icons. It is a cons of the maximum width and the maximum height. The default value is (85 . 85), which is the "correct" value for the icons defined in this utility.

ICON.SPACING [Variable]

ICON.SPACING specifies the gap between icons, for use in computing the grid positions of icons. It is a cons of the horizontal gap and the vertical gap. The default value is (5 . 5).

GRID.OFFSET [Variable]

GRID.OFFSET specifies origin of the icon grid. The default value is (0 . 0).

DEFAULTICONFONT [Variable]

The value of DEFAULTICONFONT is the default font used by the system when printing titles in icons. Since the icons defined in GRID-ICONS tend to be smaller then the original icons, you might want to use a slightly smaller font then the default. Personally, I recommend setting DEFAULTICONFONT to (FONTCREATE ’(HELVETICA 8)).

Hanoi

By: Larry Masinter (Masinter.pa@Xerox.com)

INTRODUCTION

Ancient graphics demo, upgraded to be idle hack. Adds Hanoi to list of idle displays.

OPERATION

(HANOI *NRINGS WINDOW FONT ONCE*) [Function]

Will display in *WINDOW* (or HANOIWINDOW, created first time) a towers-of-hanoi problem and solve it. It periodically blocks so you can run it as a background process. *NRINGS* is the number of rings. If *NRINGS* is a list it is the labels printed on the rings in font *FONT*. It conforms to the window shape if you reshape it. It will run indefinitely unless *ONCE* is non-NIL.

HARDCOPY-RETAIN

By: Herb Jellinek (Jellinek.pa@Xerox.com)

This document was last edited on December 29, 1987.

Introduction

This module adds a "To a file and a printer" option to the Hardcopy item on the system window and background menus. This option sends the contents of a window or screen region to a printer, and simultaneously creates a file containing a printer-ready copy of it (e.g., an Interpress master); this makes it easy to run off multiple copies later.

Using it

When loaded, HARDCOPY-RETAIN calls the function xcl-user::install-option to side effect il:|WindowMenuItems| and il:|BackgroundMenuItems| and to set il:|WindowMenu| and il:|BackgroundMenu| to nil, thus forcing them to be recalculated.

HASHBUFFER

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: HASH

HASHBUFFER combines *hash files* with *hash arrays* in order to improve hash file performance when keys are accessed multiple times. This module also defines two functions for moving data between hash files and hash arrays.

The functions below are used in place of the hash file routines. When a hash file is opened, a hash array is created, of a complimentary size. When requests for keys are made, the array is searched, and if a value is found, it is returned. If a value is not found, the file is searched and if a value is found there, it is stored in the array and returned. If a value is not found, a marker is put in the array so that the file is not searched again.

(OPENHASHBUFFER *FILE [ACCESS MINKEYS OVERFLOW HASHBITSFN EQUIVFN]*) [Function]

Opens an existing hash file and returns a hash buffer datum which must be given to the other hash buffer functions. Only the *FILE* argument is required; the *MINKEYS* argument is used for the size of the hash array and if not supplied the size of the hash file is used. Setting *MINKEYS* smaller than the size of the hash file allows a fast, small hash array window onto a larger, slower hash file. The *OVERFLOW*, *HASHBITSFN* and *EQUIVFN* arguments are passed to HASHARRAY.

(CREATEHASHBUFFER *FILE [VALUETYPE ITEMLENGTH #ENTRIESˆ*
OVERFLOW HASHBITSFN EQUIVFN]) [Function]

Like OPENHASHBUFFER but creates a new hash file. The *FILE*, *VALUETYPE* and *ITEMLENGTH* arguments are passed to CREATEHASHFILE; the *OVERFLOW*, *HASHBITSFN* and *EQUIVFN* arguments are passed to HASHARRAY. The *#ENTRIES* argument is used for both the fIle and array.

(CLOSEHASHBUFFER *HASHBUFFER [FILEONLY?]*) [Function]

Closes the hash file and sets the hash array to NIL so that it can be reclaimed. If *FILEONLY?* is non-NIL then only the hash file is closed, the hash array will be left alone.

(GETHASHBUFFER *KEY HASHBUFFER*) [Function]

(PUTHASHBUFFER *KEY VALUE HASHBUFFER*) [Function]

Retrieve and store *VALUE* for *KEY* in the hash buffer. If the hash file is only open for input, then storing a key will only affect the hash array. If the hash file is open for output, then storing a key will put it in both the hash array and hash file. If *VALUE* is NIL, then a delete is performed.

(HASHARRAY.TO.HASHFILE *HASHARRAY HASHFILE [TESTFN]*) [Function]

Uses MAPHASH to move the contents of *HASHARRY* into a hash file. If *HASHFILE* is a file name, CREATEHASHFILE is called; if *HASHFILE* is an open hash file datum, it is used and left open. *TESTFN*, if supplied, is called before each PUTHASHFILE on (KEY VALUE HASHARRAY HASHFILE) and if it returns non-NIL, the key and value are copied to the file.

(HASHFILE.TO.HASHARRAY *HASHFILE [HASHARRAY TESTFN]*) [Function]

Uses MAPHASHFILE to move the contents of *HASHFILE* into a hash array. *If HASHARRAY* is not supplied a new hash array is created. *TESTFN* is called before each PUTHASH on (KEY VALUE HASHFILE HASHARRAY) and if it returns non-NIL, the key and value are copied to the array.

HASHDATUM

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: HASH

HASHDATUM facilitates storing random Envos Lisp datatypes on *hash files* using the hashed text feature of the HASH Lisp Library module. The module defines functions which access an item on a hash file as a stream of bytes using user supplied input and output functions. Since the items are stored using text hashing, when rehashing or copying of the file occurs, the data portion of the file is copied correctly.

(GETHASHDATUM *KEY HASHFILE READFN*) [Function]

(PUTHASHDATUM *KEY DATUM HASHFILE PRINTFN*) [Function]

Use *READFN* and *PRINTFN* to store and retrieve *DATUM* on *HASHFILE*. The *READFN* takes a stream as its argument, the *PRINTFN* takes the *DATUM* and a stream. The *put* function returns the hash file text pointer record which contains two byte pointers that indicate where the datum begins and ends on the file. The *get* function returns the result of the *READFN*.

The following macros and functions are also defined using the above functions:

(GETHASHGRAPH *KEY HASHFILE*) [Macro]

(PUTHASHGRAPH *KEY GRAPH HASHFILE*) [Macro]

Use GRAPHER functions READGRAPH and DUMPGRAPH to store *GRAPH* on *HASHFILE* under *KEY*.

(GETHASHBITMAP *KEY HASHFILE*) [Macro]

(PUTHASHBITMAP *KEY BITMAP HASHFILE*) [Macro]

Use READBITMAP and PRINTBITMAP to store *BITMAP* on *HASHFILE* in a text format.

(GETHASHBINARYBITMAP *KEY HASHFILE*) [Macro]

(PUTHASHBINARYBITMAP *KEY BITMAP HASHFILE*) [Macro]

Use READBM and WRITEBM from BITMAPFNS to store *BITMAP* on *HASHFILE* in a binary format.

(GETHASHTEDIT *KEY HASHFILE [WINDOW PROPS]*) [Function]

(PUTHASHTEDIT *KEY TEXTOBJ HASHFILE*) [Macro]

Use OPENTEXTSTREAM and TEDIT.PUT.PCTB from TEDIT to store *TEXTOBJ* on *HASHFILE*, preserving both the text and formatting information. *WINDOW* and *PROPS* are optional and are passed to OPENTEXTSTREAM. If the *WINDOW* argument is not supplied, the result of the *get* function can be passed to OPENTEXTSTREAM along with a window to display the text.

(GETHASHUGLY *KEY HASHFILE*) [Macro]

(PUTHASHUGLY *KEY UGLYVAR HASHFILE*) [Macro]

Use HREAD and HPRINT to store random data, like menus, on *HASHFILE*.

HEADLINE

By: D. Austin Henderson, Jr. (AHenderson.pa@Xerox.com)

Last revised: April 1, 1986

HEADLINE contains functions for creating and closing windows which contain headlines ("headline windows").

(HEADLINE *PHRASE FONT POSITION ALIGNMENT*) [Function]

Creates a headline window with *PHRASE* printed in font *FONT* at position *POSITION* aligned as per *ALIGNMENT*; the window is just large enough to hold the headline. *PHRASE* is any Lisp object. *FONT* defines a font as per FONTCREATE (eg. (TIMESROMAN 18 BOLD)); if NIL, TimesromanD 36 is used. *POSITION* is a position giving the reference point for placing the window; if NIL, the user is given a chance to position the window with MOVEW. If *POSITION* is given, *ALIGNMENT* gives the alignment of the window with respect to *POSITION* as (xalignment . yalignment) where xalignment is one of LEFT, CENTER, or RIGHT and yalignment is one of BOTTOM, CENTER, or TOP; for convenience, if Position is CENTER then it is taken to mean (CENTER . CENTER), etc.

(HEADLINE.ARRAY *TITLES ALIGNMENT SEPARATION POSITION*) [Function]

Creates a set of vertically arranged headline windows. *TITLES* is a list of (phrase font) sublists where phrase and font are as in Headline. *ALIGNMENT* is one of LEFT, CENTER, or RIGHT, indicating how the windows are aligned with each other; defaults to CENTER. *SEPARATION* indicates the spacing between the bottoms of the windows; defaults to 70. *POSITION* indicates where the top (first) of the windows is to appear; defaults to somewhere near the top center of the screen.

(BILLBOARD) [Function]

Identical to HEADLINE.ARRAY, left in for backward compatibility.

(BANNER *PHRASE FONT POSITION ALIGNMENT*) [Function]

Same as HEADLINE except it prints the phrase vertically.

(BANNER.ARRAY *TITLES ALIGNMENT SEPARATION POSITION*) [Function]

Same as HEADLINE.ARRAY except it prints the phrases vertically, left to right.

(CLOSE.HEADLINES) [Function]

Closes all the active headline windows.

HELPSYS

By: Doug Cutting (Cutting.PA@Xerox.COM)

Uses: DINFO, HASH

This document last edited on October 7, 1987.

INTRODUCTION

HelpSys is the interface to the online version of the *Interlisp-D Reference Manual*. It provides both sequential perusal of the manual and random access lookup and display of index entries.

Interlisp-D Reference Manual DInfo Graph

Helpsys uses the **DInfo** library package as a means of accessing the *Interlisp-D Reference Manual*. Once you have loaded Helpsys, selecting **DInfo** from the **Background Menu** will raise a menu which will contain an item named **Interlisp-D Reference Manual**. Selecting this item will start DInfo on the DInfo graph for the manual. (Note: **IRM.HOST&DIR** must be set correctly before HelpSys will work; see ***Installing HelpSys***, below). See the documentation for DInfo for more information on this package.

?<cr> [EXEC macro]

Helpsys enables the EXEC and TEXEC macro **?<cr>**. Typing this (a question-mark followed by a carriage-return) will cause the CAR of the form currently being typed to be looked up. This works much like the more familiar **?=<cr>** macro which displays arguement lists.

Lookup! [DInfo Command]

Selecting Lookup! from DInfo's menu in the **IRM DInfo Graph** will prompt for and look up a term.

INSTALLING HELPSYS

Helpsys requires a number of files which should be provided with the LispUsers release. These are the 31 chapter files (CHAP*.TEDIT) the top node file (IRMTOP.TEDIT), the file containing the DINFOGRAPH (IRM.DINFOGRAPH) and the index hash file (IRM.HASHFILE).

To install HELPSYS you must copy all these files to one directory, and set the variable IRM.HOST&DIR to the name of this directory.

IRM.HOST&DIR [Variable]

This determines where HelpSys will look for the manual files and hash file. This should be set in your site init file. As file servers can often be quite loaded down, HelpSys will work much faster if you cache these files on the local disk.

IRM.HASHFILE.NAME [Variable]

If this is NIL, HelpSys will look for the hash file as IRM.HASHFILE on IRM.HOST&DIR, otherwise it uses the value of this variable. Note that the hash file *must* be on a random access filing device (eg. the local disk).

PROGRAMMERS' INTERFACE

(IRM.LOOKUP *KEYWORD TYPE GRAPH SMARTFLG*) [Function]

This is the primary function of HelpSys. If *TYPE* is specified, then the primary manual entry for *KEYWORD* of *TYPE,* or the first if there is no primary, will be displayed in *WINDOW.* If *TYPE* is not specified, then a pop-up menu will be raised containing all the manual entries for *KEYWORD*. Primary entries are marked with stars (*'s) on either end of the item. *GRAPH* should be the IRM Dinfo graph, and defaults to IRM.DINFOGRAPH.

(IRM.SMART.LOOKUP *KEYWORD GRAPH*) [Function]

Uses wild card matching if *'s are in *KEYWORD* (* matches any substring) or tries spelling correction. A pop-up menu is raised if more than one wild card match is found. Note that the first time a * appears in a *KEYWORD,* HelpSys will need to load the list of possible keywords for matching against, and only after this list has been loaded will spelling correction be enabled. This is the function called by the Lookup! button in DInfo's FreeMenu. *GRAPH* again defaults to IRM.DINFOGRAPH.

(IRM.RESET) [Function]

Will reset HelpSys so that everything will be reinitialized.

H

By: Roberto Ghislanzoni (Roberto Ghislanzoni:MKT:RXI)

INTRODUCTION

The package H allows the user to augment the Interlisp–D environment with an Horn Clauses Theorem prover: in it it's possible to call semantic attachments (SAs) to Lisp (i.e. lisp functions) as FOL does.

A BRIEF HISTORY

The original idea comes from Chester (Chester,1980), (Chester, 1980): it was revised by Simmons (Simmons, 1984). A prototype of this program was developed at the University of Milan by Vieri Samek Ludovici, Giorgio Tornielli and Roberto Ghislanzoni using VLisp. Currently it is offered on Interlisp–D environment.

USE OF PACKAGE

Load the file H-LOAD.DCOM. This loads all necessary files. On your machine, you have now two environments: the H developer and the H deliver. In H developer you can plan, construct and edit your HKBs (H Knowledge Bases), and use it from Lisp executive window, simply by H.loading the HKBs created. H is a very good logic paradigm for LOOPS: from executive window, it is possible to have as many calls to the prover as you need.

THE H DEVELOPER

From the background menu chose H: this offer you a window in which is active a *read-prove-print* loop. Open as many windows as you want: all HKBs are local to the windows. The H Control Window has the following entries:

— **Show Profile** : This shows in the Prompt Window the current settings for environment bound to the window; the MODE of demonstration (FIRST: stop to the first goal proved; ALL: reach all goals: T: interactive mode); the LIMIT of the search tree; the TRACING of prover: the PMTRACING, that shows the pattern matching at work.

— **Show(Axiom)** : shows the clauses that define a predicate; the submenu allows to see the lambda-definition of a semantic attachment. The choice is made from a pop-up menu.

— **Delete(Axiom)** : this erases from database the clauses choosen by the user. Erases also the SA from the submenu.

— **Edit(Axiom)** : it allows to create and edit both predicates and SA, using the standard DEdit facilities.

— **SetLimit** : sets the limit of the search tree.

— **Mode** : chose the mode of demonstration.

— **Shortform** : it enables or disabled the control for occurrence, so it is not possible to unify the variables already bound in that piece of unification to themselves.

— **Trace** : it enables in a separate window the tracing of demonstration.

— **Trace PM** : it enables the tracing of the unifier.

— **LoadHKB** : loads a H Knowledge base in the environment: the name of KB is shown beside the window; do not provide the .HKB extension to the name: the system does it.

— **SaveHKB** : saves the current environment in a KB: don't provide extension.

— **EraseEnv** : erases the entire environment.

— **Exit** : exits and closes window.

All the windows that H uses ("Show window", "Trace window", "PM Trace window") has the middle button capability in order to allow to dribble into a file everything that is printed in the window; it is very similar to CHAT's dribble option.

THE H DELIVER

There are a lot of functions available from Lisp Executive that allow the programmer to use the HKBs previously created:

(H.erase) [Function]

Erases all environment (i.e., predicates and SAs) previously loaded.

(H.load *database*) [Function]

Loads H database into environment.

(H.save *database*) [Function]

Saves all current environment into a file

(H.? *pred1 ... predN*) [NLAMBDA Function]

Start the demonstration of the predicates specified. Return the list of predicates proved with the variablesof the call set .

(H.all *variables conjs*) [Function]

Returns the list of all specified variables that satisfy the predicate(s). Remember that variables must begin with a ':' (semicolon).

(H.any *howmany variables conj*) [Function]

Returns *howmany* instantiation values of variables that make true the predicate(s).

(H.attach *foo lambda–expression*) [NLAMBDA Function]

Defines a SA named *foo* to be the value of the LAMBDA-expression written as the second argument.

(H.addaxiom *axioms–list*) [Function]

Adds new axioms to the existing one for that predicate.

(H.axiom *axioms–list*) [Function]

Defines new axioms for the predicate. Deletes the previous ones.

(H.del *axiom*) [Function]

Deletes a single axiom from the database; the other axioms for that predicates are not touched.

(H.show) [NLAMBDA Function]

Shows the definition of the given predicates.

(H.the *variable conjs*) [Function]

Returns only one value for the variable that satisfies the goal.

(SET.H.MODE mode num) [Function]

Set the mode of demonstration: mode may be one of atoms 'first, 'all, 'interactive. If atom 'limit, then you must provide the number of depth (default 200).

H PRIMITIVES

Both environments have three important primitives SA:

(set *var expr*)

Sets in the current level of unification the variable to the value of the expression expr.

(assert *axiom*)

Assert in the database the given axiom. without erasing the old ones.

(delete *axiom*)

Delete in the database the given axiom.

In the system also is present the cut facility ('/'), that has similar behaviour as PROLOG cut ('!').

H DEMOS

An example of axioms may be this:

```
(((append () :a :a))
 ((append (:a . :b) :c  (:a . :d)) < (append :b :c :d)))
```

You can call from H-executive:

```
 (append (1 2 3) (4 5 6) :d)
```

that returns

```
 ((append (1 2 3) (4 5 6)  (1 2 3 4 5 6))
```

Also, if you have in your database:

```
(((A 1))
 ((A 2)))
```

and

```
(((B 3))
 ((B 4)))
```

you can call from TTY exec:

```
(H.? (A :l))  --> (((A 1)))
```

or

```
(H.? (A :k) (B :o))  -->  (((A 1) (B 3)))
```

Moreover:

```
(H.all ':k '((A :k)))  --> (2 1)
(H.all '(:a :b) '((A :a) (B :b)))  --> ((2 4) (2 3) (1 4) (1 3))
(H.any 2 ':j '((A :j)))  --> (2 1)
```

For demo. load the HKBs H-MAZE,H- BLOCKS and try the following:

```
(showworld)
```

that shows you the block world situation: try then

```
(please (put the red cube on the blue one))
 (please (pick up cube2))
```

and so on

The other examples solve for you the maze problem and other interesting things: discover by yourself how they work ...

REFERENCES

Chester D., Using HCPRVR, Department of Computer Sciences, University of Texas, Austin, June 1980

Chester D, HCPRVR: a logic program interpreter in Lisp, proc AAAI, Department of Computer Sciences, University of Texas, Austin, June 1980

Simmons R.F.,Computaions from the English, Prentice Hall, New Jersey, 1984

HISTMENU

Original Fugue version By: Danny Bobrow (Bobrow.PA@Xerox.COM)

2020 Medley 3.5 clean sheet reimplemnentation By: Michele Denber (mdenber@gmail.com)

INTRODUCTION

HISTMENU is a Xerox Lisp (Medley, Lyric or Koto) program that provides quick access to commands recently typed in the Exec window. The original HISTMENU was written by Danny Bobrow but seems to have been lost over time. This version was reverse engineered from a running instance in an old sysout.

OPERATION

Load HISTMENU.LCOM from your local Lispusers directory.   Then call

(HistoryMenu [histMenuLength] [histMenuPosition])

where

histMenuLength optionally specifies the number of commands you want displayed. Default is 30.

hitMenuPosition optionally specifies a position on the display to place the menu. Default is to place it using the mouse.

Clicking Left on any menu item will redo that command.

Clicking Middle brings up a pop-up menu that lets you issue one of the four Programmer's Assistant commands REDO, FIX, UNDO, ?? plus an option Delete which removes that item from the History Menu.

Clicking Right bring up a pop-up menu with the three standard options Bury, Move, and Shrink plus an option Update which updates the entries in the History Menu to again show the last n commands from the Exec window.

I have made one change to the original HISTMENU. In the original, it would send the command to whatever window had the keyboard focus. So if you were in a TEdit window, clicking a HISTMENU itwm would place "REDO 354" in the TEdit window. This version only allows commands to go into a selected Exec window.

OPTIONS

HPGL

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

HPGL defines a Lisp *image stream* type that generates output for plotters (and other devices) which use the *Hewlett-Packard Graphics Language*. The module does not define any user functions, the HPGL streams are accessed via OPENIMAGESTREAM and the hardcopy functions.

PLOTTERS

Some plotters which use HPGL either as their primary language or as an optional extra:

Hewlett-Packard (most)	Epson America HI-80 (option)
Facit 4551	Western Graphtec MP1000 and FP5301 (option)
Gould Color-writer 6120 and 6320	Houston Instrument DMP-29 (option)
Taxan 710	Nicolet Zeta 8 (option)
IBM 7371	
Roland DYX-880 and 980	(source: *PC*, Volume 4, Number 26, December 1984)

The file extensions HPGL and PLOT are recognized by the system as plotter output file types.

OPTIONS

The driver accepts the following in the OPTIONS argument to OPENIMAGESTREAM:

SCALE	Image scaling; value should be a POSITION record which indicates where the second scaling point should be placed (the initial scaling point is at 0,0). By default, uses `(SCREENWIDTH . SCREENHEIGHT)`.
ROTATE	Paper rotation; value should be 0 or 90, defaults to *landscape* plotting (0).
PAPER	Paper size; value is a small integer, the HP 7475A accepts 3 (A3) or 4 (A4).
TERMINATOR	Label terminator character; value should be a character, the default is '^A'.
VELOCITY	Pen velocity; plotter specific.

IMPLEMENTATION

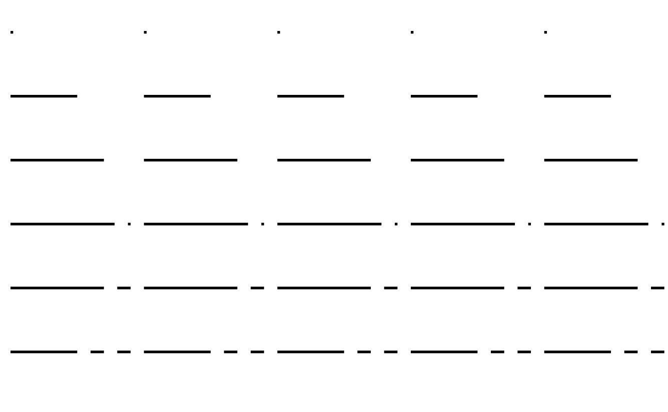
The driver was implemented using an HP 7475A plotter but the plotter output conforms to the more restrictive HP 9872 syntax to be more widely applicable. The driver uses the following variables which may need adjustment for other types of plotters:

HPGL.FONTS	An ALST of font names and (small integer) plotter font numbers.
HPGL.OPTIONS	An ALST of plotter specific options that can be passed to OPENIMAGESTREAM and the corresponding HPGL command to print.
HPGL.DASHING	An ALST of HPGL line types (small integers) and dashing lists.
HPGL.FONT.EXPANSIONS	An ALST of font face expansions (REGULAR, COMPRESSED and EXPANDED) and the relative scale of each.
HPGL.TERMINATOR	The default end of instruction terminator character, initially ';'.

HPGL.SEPARATOR	The default parameter separator character, initially ‘,’.
HPGL.TEXT.TERMINATOR	The default end of label terminator character, initially ‘^A’.
HPGL.CHORD.ANGLE	The chord angle used by the circle and arc instructions. Defaults to NIL which causes the plotter’s default to be used.
HPGL.PATTERN.LENGTH	The default pattern length for the hardware dashing. Defaults to NIL which causes the plotter’s default to be used.
COLORNAMES	System variable used to convert between RGB triples and pen numbers. The order of entries affects the pen number to color correspondences.

DASHING

To minimize the complexity of the driver and maximize the speed of plotting, for operations other than DRAWLINE, the driver only uses the built-in dashing types of the plotter. The correspondences between the dashing style and HPGL line type number are kept in the HPGL.DASHING variable which can be modified or extended for plotters with different dashing styles than those displayed below:



1	(1 49)
2	(25)
3	(35 15)
4	(39 5 1 5)
5	(35 5 5 5)
6	(25 5 5 5 5 5)
	NIL

If the driver is loaded after SKETCH, the dashing types are added to SKETCH’s dashing menu .

IDEASKETCH

By: Richard Burton (Burton.Pa@Xerox.com)

Uses: Sketch

This document last edited on 26-Jan-89 19:09:56.

INTRODUCTION

Idea Sketch is an adaptation of Sketch that is designed to allow easy jotting down and laying out of ideas. An idea sketch window is actually a sketch window with a command menu tuned for manipulating text and defaults set up for connecting text (i.e. arrowheads are added.)

Selecting the "More Menu" items gets the standard sketch menu.

IDLEHAX

By: Larry Masinter (Masinter.pa@Xerox.com) with contributions by various others.

INTRODUCTION

This module contains a couple of random demonstration programs, useful as "Idle programs", callable from the background menu. The Idle display options includes Lines Warp-Out Radar Triangles RandAngles Polygons Bubbles and Kaleidoscope.

These are implemented by the following functions:

(POLYGONS *W NOBLOCK TIMER*) [Function]

Calls (POLYGONS) or (POLYGONS window) to perpetually draw polygons in the given window (it (re)uses POLYGONSWINDOW if argument is NIL). To run in the background, you can ADD.PROCESS((POLYGONS (CREATEW]. Controlled somewhat by the global parameters POLYGONMINPTS (minimum number of vertices), POLYGONMAXPTS (maximum number of vertices), POLYGONSTEPS (number of steps between min and max), and delays POLYGONWAIT (time between different polygons) and POLYGONWAIT2 (delay between initial display of beginning and end and the movement phase.)

If NOBLOCK is T, it doesn't block at all (runs after but can't run in background.) If TIMER is given, then POLYGONS will stop after TIMER is expired. (Used by the demo system.)

(LINES *W N LCNT STEPS ODDSTEP*) [Function]

Similar to POLYGONS in controls, but draws perpetually changing form using line draw. W defaults a "demo window", but is the window on which the display is drawn, N is the number of endpoints (e.g., 2 draws lines, 3 draws triangles, 7 draws 7-segment figures), LCNT is the "number of lines on the screen at any one time", STEPS is the number of lines to draw between start and end (the higher this number, the closer together the lines are), and ODDSTEP is a flag: if T, then the odd endpoints remain the same every other iteration (try (LINES NIL 3 1 40 T).) The background RandAngles means: (LINES W (RAND 3 7) (RAND 1 16) (RAND 25 100)), while Triangles is (LAMBDA (W) (LINES W 3 1 40)), etc.

(BUBBLES *WINDOW*) [Function]

Perpetually draws circles. Controlled by BUBBLECNT, which is read at startup as the number of circles visible at any one time.

(KAL *W PERIOD PERSISTENCE*) [Function]

Borrowed from the KAL LispUsers package: draws a random symmetric pattern of dots. Pretty. Period affects the style of display, while PERSISTENCE affects how many dots are on the screen at once.

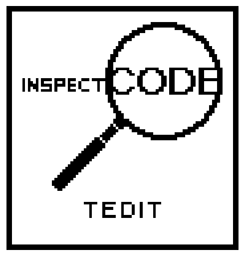
(WARP *W*) [Function]

Draws a sequence of circular patterns that resemble piles of sand. Or not; you decide.

POLYGONS, LINES and BUBBLES adjust themselves to the size of the window, so you can reshape the window in the middle of the demo.

INSPECTCODE-TEDIT

By: Matt Heffron (BEC.HEFFRON@ECLA.USC.EDU)
Beckman Instruments, 2500 Harbor X-11
Fullerton, CA. 92634
(714) 961-3128



The **INSPECTCODE-TEDIT** package advises the INSPECTCODE facility to have some extended capabilities when the TEDIT and GRAPHCALLS packages are loaded (i.e. it uses TEDIT and GRAPHCALLS).

If TEDIT is not defined, then the standard INSPECTCODE will be used. If TEDIT is loaded, then a read-only TEDIT/INSPECTCODE window will be opened, and will have a special INSPECTCODE menu for **LEFT** or **MIDDLE** buttoning in the titlebar. All of the options, except for **Quit**, in this menu use the current selection in the window. You make selections with the mouse buttons in the standard TEDIT ways. The options in the INSPECTCODE titlebar menu are:

GraphCalls If the GRAPHCALLS package is loaded, then calls **GRAPHCALLS** on the current selection.

InspectCode Opens a new **INSPECTCODE** window on the current selection.

Inspect Does an **INSPECT** on the value of the current selection. This item has SUBITEMS (see below).

Pretty Print Value Prompts for region to open a window, and prettyprints the value of the current selection in it. This item has SUBITEMS (see below).

Quit Closes this window and kills the associated TEDIT process. (Closing the window with the WindowMenu, or by calling **CLOSEW** on it does the same thing.)

The **Inspect** and **Pretty Print Value** menu options have the following SUBITEMS which affect how the value of the current selection is determined:

Freely The value of the current selection is determined by any binding that a free-reference from the INSPECTCODE window menu handling code (i.e by

(**EVALV** selection)). This is the default behavior when a menu selection is made directly from the titlebar menu without using the SUBITEMS menu.

Globally The value of the current selection is determined by its top level (Global) binding.

In Process Context The value of the current selection is determined by its binding in the context of a specified process. A menu of all current processes will be brought up to allow you to specify a process.

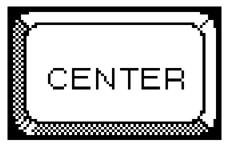
INSPECTCODE-TEDIT also defines the LISPXMACRO **IC** which INSPECTCODE's its argument.

KEYOBJ

By: Greg Nuyens

Supported by Jan Pedersen (Pedersen.pa@Xerox.com)

KEYOBJ provides a LISP imageobject which mimics a key. The default image looks like this:



These keys are pressed by clicking the mouse inside the key's image. The result of pressing a key is determined (just like the physical key) by the Interlisp-D system function KEYACTION. To enter a KEYOBJ into TEdit type ^o. Inside the window that pops up, call the following function:

(KEYOBJ.CREATE KeyName KeyLabel Abortable) [Function]

KeyName is the key that you want the object to behave like. (CENTER in the example above). KeyLabel is an optional label other than the key whose action it mimics. If KeyLabel is a list of two elements, the first is displayed above the second. Abortable is a flag which indicates that no transitions should be generated if the mouse button is released outside the key image.

KEYOBJ.FONT [Variable]

Determines the font in which the label is created inside the keyobj. Default is Helvetica 10.

KINETIC

By: Anon.

Recompiled for Medley by Larry Masinter (Masinter.PA@Xerox.COM)

INTRODUCTION

An ancient graphics hack, converted to work with idle.

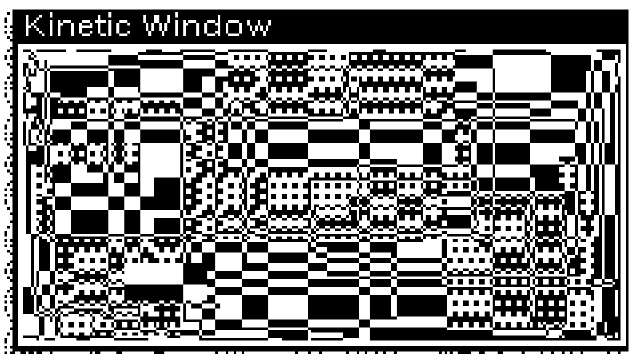
OPERATION

(KINETIC *WINDOW*) [Function]

to randomly invert rectangles on *WINDOW*, or on KINETICWINDOW (set up first time). Choosing the Kinetic on the Idle Choose Display menu will select the KINETIC function as the Idle display.

CHECKSHADE [Variable]

If non-nil, CHECKSHADE is a texture which is used for some of the rectangles sometimes. Defaults to 63903.



LAMBDATRAN

By: >>Your Name<< (>>Your net address<<)

>>Other packages necessary to run this one<<

The purpose of this package is to facilitate defining new LAMBDA words in such a way that a variety of other system packages will respond to them appropriately. A LAMBDA word is a word that can appear as the CAR of a function definition, like LAMBDA and NLAMBDA. New LAMBDA words are useful because they enable the user to define his or her own conventions about such things as the interpretation of arguments, and to build in certain defaults about how values are returned. For example, the DECL package defines DLAMBDA as a new LAMBDA word with unconventional arguments such as the following:

```
(DLAMBDA ((A FLOATP) (B FIXP) (RETURNS SMALLP))

(FOO A B))
```

In order for such an expression to be executable and compilable, a mechanism must be provided for translating this expression to an ordinary LAMBDA or NLAMBDA, with the special behavior associated with the arguments built into the function body. The LambdaTran package accomplishes this via an appropriate entry on DWIMUSERFORMS that computes the translation.

Besides executing and compiling, Interlisp applies a number of other operations to function definitions (e.g., breaking, advising), many of which depend on the system being able to determine certain properties of the function, such as the names of its arguments, their number, and the type of the function (EXPR, FEXPR, etc.). The LambdaTran package also provides new definitions for the functions FNTYP, ARGLST, NARGS, and ARGTYPE which can be told how to compute properties for the user's LAMBDA-words.

A new LAMBDA-word is defined in the following way:

1. Add the LAMBDA-word itself (e.g., the atom DLAMBDA) to the list LAMBDASPLST. This suppresses attempts to correct the spelling of the LAMBDA-word.

2. Add an entry for the LAMBDA-word to the association list LAMBDATRANFNS, which is a list of elements of the form: (*LAMBDA-WORD TRANFN FNTYP ARGLIST*), where (*LAMBDA-WORD* is the name of the LAMBDA-word (e.g., DLAMBDA).

TRANFN is a function of one argument that will be called whenever a real definition is needed for the LAMBDA-word definition. Its argument is the LAMBDA-word definition, and its value should be a conventional LAMBDA or NLAMBDA expression which will become the translation of the Lisp LAMBDA-word form. The free variable FAULTFN is bound to the name of the function in which the LAMBDA-word form appeared (or TYPE-IN if the form was typed in).

FNTYP determines the function type of a definition beginning with LAMBDA-WORD. It is consulted if the definition does not already have a translation from which the function type may be deduced. If *FNTYP* is one of the atoms EXPR, FEXPR, EXPR*, or FEXPR*, then all definitions beginning with LAMBDA-word are assumed to have that type. Otherwise, *FNTYP* is a function of one argument that will be applied to the LAMBDA-word definition. Its value should be one of the above four function types.

ARGLIST determines the argument list of the definition if it has not already been translated (if it has, the *ARGLIST* is simply the *ARGLIST* of the translation). It is also a function of one argument, the LAMBDA-word definition, and its value should be the list of arguments for the function (e.g., (A B) in the DLAMBDA example above). If the LAMBDA-word definition is ill formed and the argument list cannot be computed, the function should return T. If an *ARGLIST* entry is not provided in the LAMBDATRANFNS element, then the argument list defaults to the second element of the definition.

As an example, the LAMBDATRANFNS entry for DLAMBDA is (DLAMBDA DECL EXPR DLAMARGLIST), where DECL and DLAMARGLIST are functions of one argument.

Note: if the LAMBDA-word definition has an argument list with argument names appearing either as literal atoms or as the first element of a list, the user should also put the property INFO with value BINDS on the property list of the LAMBDA-word in order to inform DWIMIFY to take notice of the names of the arguments when DWIMIFYing.

LIFE

By various folks, including help from Mike Dixon (MikeDixon.PA@Xerox.COM) and Larry Masinter (Masinter.pa@Xerox.com)

This Life program is a translation of the Smalltalk-80 version in the book **Smalltalk-80: The Language and its Implementation**, by Goldberg and Robson.

Input is a window where the "on" pixels are interpreted as living cells. The window is continually updated as life goes on.

Now an "idle" hack: LIFEDEMO as a display function plays life with the bits of the screen (in a copy of them in a window, e.g., it doesn't smash your screen.)

(LifeIdle W N) [Function]

Run Life in window W, using the bits behind W as a starting point. N is optional, and can either be 1, 2,4 or 8. Its the magnification of the life window.

(Life W N) [Function]

Like LifeIdle but uses the current contents of the window.

LOADMENUITEMS

By: sML (Lanning.pa@Xerox.com)

INTRODUCTION

Some utility files are so useful that users will always want them in their system: these files are typically loaded from the users INIT file. A (rather large) number of other utilities are only sometimes useful. Users are faced with the choice of either loading these files from their INIT files (slowing down the initialization process and consuming space, whether the utility is needed or not) or having to remember how to load and initialize these files.

LOADMENUITEMS addresses this problem: it defines a new filepackage command that can be used to add entries onto the background menu for easy loading of utility files.

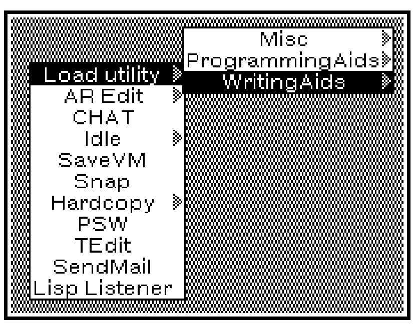
[NOTE: All (advertised) symbols in this utility are in the INTERLISP package.]

EXAMPLE

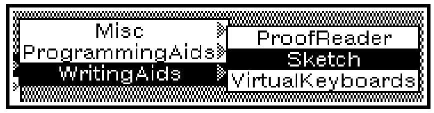
The filepackage command

```
(COMS
      ;; Make it easy to load some oft-used utilities
      (FILES LoadMenuItems)
      (LOADMENUITEMS WritingAids Sketch VirtualKeyboards ProofReader)
      (LOADMENUITEMS ProgrammingAids (Spy (SPY.BUTTON)))
      (LOADMENUITEMS NIL VStats Calendar))
```

will add an entry "`Load utility`" to the background menu. "`Load utility`" will have three subitems: `Misc`, `ProgrammingAids`, and `WritingAids`:



`WritingAids` will in turn have three subitems: `ProofReader`, `Sketch`, and `VirtualKeyboards`; `Misc` will have the subitems `Calendar` and `VStats`; `ProgrammingAids` will have the single subitem `Spy`.



Selecting any of these final menu items will load the corresponding file. In addition, the `Spy` menu item will evaluate the form `(SPY.BUTTON)` after loading the file Spy.

INTERFACE

(LOADMENUITEMS *group utilDescr1 utilDescr2 ...)* [FilePackageCommand]

Dumps out to the file a form that will add items to the background menu for loading *utilDescr1*, *utilDescr2*, ... Each item will be added to the *group* subitem of the "Load utility" item on the background menu; if *group* is NIL it defaults to "Misc".

In the simplest case, *utilDescr* is a LITATOM. This is used when you want to load a file without any extra initialization, and the file is on one of the directories in `DIRECTORIES`. Selecting the resulting item will evaluate (DOFILESLOAD '*utilDescr*) and print an informative message in the prompt window when the DOFILESLOAD is finished. The added item will have the label *utilDescr*.

In the general case, *utilDescr* is a list. This is used when you want to specify an initialization form to be evaluated when the utility is loaded, or when the file description is not a LITATOM. In this case, selecting the menu item will evaluate (DOFILESLOAD (CAR '*utilDescr*)). If *utilDescr* is a list of two elements, the CADR of *utilDescr* will be evaluated after the utility is loaded; otherwise an informative message will be printed in the prompt window. The added item will have as a label the first LITATOM in the CAR of *utilDescr*; this is the first file that will be loaded when the item is selected.

In each of the above cases, the item is removed from the background menu after the utility is loaded and initialized.

When a utility is loaded from the "Load utility" menu, the event is added to the history list. This way you can UNDO loading a utility.

Some illustrative examples:

```
                              ;; This adds the item "VStats" to the "Misc" subitem
(LOADMENUITEMS NIL VStats)

                              ;; Selecting the "Spy" item will load SPY and call
                                 SPY.BUTTON to bring up the spy button icon
(LOADMENUITEMS ProgrammingAids (Spy (SPY.BUTTON)))

                              ;; This will add the item "GO" to the "Games" group
(LOADMENUITEMS Games (((SYSLOAD FROM {PHYLUM}<Foster>Lisp>) GO)))

                              ;; These items are useful for Lafite users, but aren't always
                                 needed
(LOADMENUITEMS MailTools LafiteFind Undigestify MailScavenge)
```

FUNCTIONS

(AddLoadMenuItem *group fileDescr startUpForm*) [Function]

Add a menu item to the background menu that will load the files. The item will be added under the top level item "Load utility". *group* is the submenu name for this file; the default is `Misc`. *fileDescr* is a list that can be passed to DOFILESLOAD to load the files. *startUpForm* is an optional form that will be evaluated after the DOFILESLOAD; the default will print a nice message in the prompt window. The LOADMENUITEMS filepackage command described above expands to calls to AddLoadMenuItem.

AddLoadMenuItem is UNDOable.

(PickLoadUtilityItem *utility-name* &OPTIONAL *group-name no-errors-p*) [Function]

This is the programatic equivilent of selecting the item named *utility-name* from the "Load utility" item on the background menu. If *group-name* is given, only that group undher the "Load utility" item is searched for the utility; otherwise the entire menu item is searched. If multiple matching items are found, a continuable error is signaled. Proceeding from this error will let you pick one of the items to execute. If no matching items are found, a continuable error is signaled. The *no-errors-p* flag controls whether or not these errors are actually signaled: if *no-errors-p* it true, PickLoadUtilityItem ignores the errors. PickLoadUtilityItem return T if the utility was loaded, NIL otherwise.

LOGIC

By: Roberto Ghislanzoni ("Roberto_Ghislanzoni".MKTRXI@Xerox.com)

Uses: TABLEBROWSER

This document last edited on 20-Dec-1988 00:52:22.

NOTES ABOUT LOGIC MEDLEY RELEASE

This LOGIC release(1.3) is more robust than previous on top of Lyric; there are some bugs fixed. The major enhancement is the possibility of handle multiple SEDIT sessions: there is a global variable, *LOGIC-CLOSE-ON-COMPLETION-FLG* (defaulting to T), that controls the behaviour of the editing: NIL means not to close the editing window and not to perform a check on the correctness of the axiom just typed in, T means to check definitions and to close the editing window.

INTRODUCTION

This package is devoted to people who want to use a logic paradigm in their programming environment. LOGIC was initially developed in Franz Lisp at the Computer Science Department of the University of Milan: now a modified part of its kernel is running in Common Lisp, so it is possible to use it under every machine running CL: within the Xerox environment, some features are available in order to ease the construction of the programs.

All of the source codes are available: sorry if they are awful! But it's better to have efficiency than syntactic sugar ...

LOGIC MANUAL

LOGIC is essentially a theorem prover based on Horn clauses: the user is allowed to create many theories and within these theories to specify some predicates (clauses); as FOL does, it is also possible to specify some *semantic attachements* (SA), in order to use all the capabilities of the environment: in our implementation, these SAs are expressed in Lisp. A goal proof is performed within specifed theory(es); the user is allowed to dinamically change the theories involved.
These are the elements of the language:
• a *variable* is an atom beginning with '?'
• an *atomic formula* is a list beginning with the name of the predicate and followed by the terms: (on table book)
(mother ?x ?y)
• a *clause* is a list beginning with the consequent, followed by the special symbol ':-', and by the sequence of the antecedents:

((grandfather ?x ?y) :- (father ?x ?z) (father ?z ?y))
• a *set of clauses* is the definition of a predicate
(((append () ?a ?a)) ((append (?a . ?b) ?c (?a . ?d)) :- (append ?b ?c ?d)))
• a *theory* is a set of the definitions of some predicates

HOW TO LOAD AND INIT LOGIC

In order to use LOGIC, load the files LOGIC and LOGIC-UNIFIER. From within the Xerox Lisp Environment, you can also load the development environment LOGIC-DEVEL.DFASL. After loading the files, call the functions:

• (CREATE-BACKGROUND-THEORY): this function creates the main theory reading the data it needs from the file LOGIC.LGC.

• (CREATE-VARIABLES): this functions creates and initializes the variables used by the unifier; it takes a few time to perform its job. This call is due to a lack of the Xerox garbage collector: since the unifier uses techniques of redenomination, a great number of symbols is generated; the 1186 does not release the space used by these symbols, and so they fill up the GC table. The workaround is to re-use all the symbols generated.

If you want to port this code on another machine running CL, it is a matter of taste to eliminate this piece of code, with a little hacking on the source codes.

These are the functions available from the top-level executive of Lisp:

(ALL *VARS CONJ THS*) [Function]

Returns the *vars* that satisfies the goal (*conj*) in the list of theories (*ths*): the background theory is always used.For example you can ask the system to prove:
(ALL '(?a ?b) '((append ?a ?b (1 2 3))) '(append-theory))
--> ((NIL (1 2 3)) ((1) (2 3)) ((1 2) (3)) ((1 2 3) NIL))

(ANY *HOW-MANY VARS CONJ THS*) [Function]

Returns *how-many vars* that satisfies the goal:
(any 2 '?a '((append ?a ?b (1 2 3))) '(append-theory))
--> (NIL (1))

(ATTACH *SA-NAME DEFINITION THEORY-NAME*) [Function]

Allows to create semantic attachments:
(ATTACH 'createw '(lambda (name) (IL:CREATEW () name)) 'my-theory)
and now:
(ANY 1 () '((createw "Kiss me on my lips")) '(my-theory)
--> ;;creates a window on the screen

(CREATE-THEORY *THEORY-NAME*) [Function]

Creates a brand-new theory with that name: return the name of the theory created, not the theory itself.

(LIST-ALL-THEORIES) [Function]

Return a list of the defined theories, currently available.

(LOAD-THEORY *THEORY-NAME*) [Function]

Loads from the current directory the specified *theory-name*; the name of the theory file has the extension .LGC , and it must be previously created by the corresponding function SAVE-THEORY

(LOGIC-ADDA *PRED CLAUSES THEORY-NAME*) [Function]

Adds to the definitions of the predicate *pred* the specified *clauses*, that holds in the theory *theory-name*: the clauses are put in front of the already existing clauses:
(LOGIC-ADDA 'C '(((C 1)) ((C ?x) :- (A ?x))) 'my-theory)

(LOGIC-ADDZ *PRED CLAUSES THEORY-NAME*) [Function]

Adds to the definitions of the predicate *pred* the specified *clauses*, that holds in the theory *theory-name*: the clauses are put at the end of the already existing clauses:
(LOGIC-ADDZ 'C '(((C 2)) ((C ?x) :- (A ?x) (B :y))) 'my-theory)

(LOGIC-ASSERT *PRED CLAUSES THEORY-NAME*) [Function]

Replaces all previous definitions of the predicate *pred* with *clauses*.

(LOGIC-DELETE *PRED-OR-SA THEORY-NAME*) [Function]

Erases from the theory *theory-name* the definition of *pred-or-sa*, that may be either a predicate or a semantic attachment

(LOGIC-DELETE-FACT *FACT-NAME FACT-CLAUSE THEORY-NAME*) [Function]

Erases from the definition of the clauses on the predicate *FACT-NAME* the specified clause *FACT-CLAUSE*, within the theory *THEORY-NAME*.

(MERGE-THEORIES *NEW-THEORY-NAME &REST LIST-OF-THEORIES)* [Function]

Creates the new theory *NEW-THEORY-NAME* made up by all the predicates and sas that hold in all the theories *LIST-OF-THEORIES*: now no control is performed on the consistency in the merging of the theories

(PROVE *CONJ THS*) [Function]

Calls the prover on the specified goals *conj*. *THS* is a list of the theory(es) used. PROVE returns only T or NIL

(SAVE-THEORY *THEORY-NAME*) [Function]

Writes on the local directory the contents of the theory *theory-name*. You will find later a file whose name is composed by the theory name and the extension LGC.
The format of the contents of the file is the following:

```
theory-name
number of semantic attachments
<sa name1> <sa definition>
..
<sa nameN> <sa definition>
number of predicates
<predicate name 1> <clauses for predicate 1>
..
<predicate name N> <clauses for predicate N>
```

A theory file (with .LGC extension) may be created by the user employing a text editor like Emacs or VI (on Symbolics, SUNs etc.), avoiding the saving of the theory every change he performs.

(SHOW-DEFINITION *ELEMENT THEORY-NAME*) [Function]

Shows the definition of *element*, either a predicate or a semantic attachment.

(SHOW-THEORY *THEORY-NAME &OPTIONAL VERBOSE*) [Function]

Shows the contents (name of predicates and sas) of the theory *theory-name*; if *verbose* is T, all the definitions are shown.

THE BACKGROUND THEORY

In the background theory, many interesting primitive predicates are available:

! [Predicate]

The cut predicate, well-known to the PROLOG programmers: a tipical example of its use can be the definition of the predicate NOT:
(((not ?formula) :- (wff ?formula) ! (fail))
((not ?formula)))

(TRUE) [Predicate]

This predicate always succeds

(FAIL) [Predicate]

The predicate that never succeds

(PRINT ?arg) [Predicate]
Prints the argument ?arg passed by

(EVAL&PRINT ?arg) [Predicate]
This predicate evaluates and print the result of evaluation of the form ?arg:
(prove '((eval&print (+ 3 4))) '(my-theory))
7
T

(LOGIC-ADDA *?PREDICATE-NAME ?CLAUSES ?THEORY-NAME*) [Predicate]
Adds in front of the clauses that define the predicate ?PREDICATE-NAME in the theory ?theory-name the other clauses ?CLAUSES

(LOGIC-ADDZ *?PREDICATE-NAME ?CLAUSES ?THEORY-NAME*) [Predicate]
Adds to the end of the clauses that define the predicate ?PREDICATE-NAME in the theory ?theory-name the other clauses ?CLAUSES

(LOGIC-ASSERT *?PREDICATE-NAME ?CLAUSES ?THEORY-NAME*) [Predicate]
Replaces all definition for the predicate ?PREDICATE-NAME in the theory ?THEORY-NAME with the new clauses ?CLAUSES

(LOGIC-DELETE *?PREDICATE-OR-SA--NAME ?THEORY-NAME*) [Predicate]
Deletes all definition for predicate (or sa) ?PREDICATE-OR-SA-NAME in the theory ?THEORY-NAME

(LOGIC-DELETE-FACT *?FACT-NAME ?FACT-CLAUSE ?THEORY-NAME*) [Function]

Erases from the definition of the clauses on the predicate *FACT-NAME* the specified clause *FACT-CLAUSE*, within the theory *THEORY-NAME*.

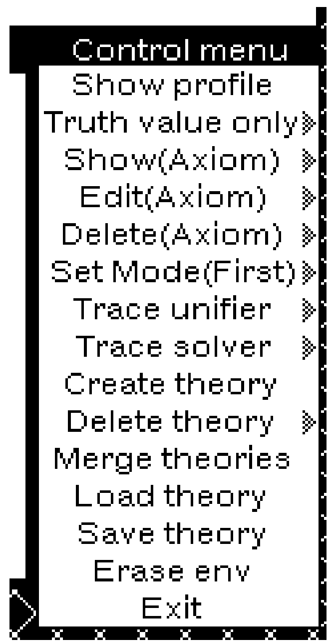
(SET ?var value) [Predicate]
With this predicate it is possible to set a variable within the demonstration (remind that a variable always starts with a '?'):
(prove '((set ?x (list 'a 'b 3))(print ?x)) '(my-theory))
--> (a b 3)
T

(RETRACT ?theory-name) [Predicate]
Tells the interpreter that it must use no more the theory *?theory-name* during the ongoing demonstration; this elision is made only on the current active node of the demonstration tree

(USE-THEORY ?theory-name) [Predicate]
Tells to the interpreter that it must use the theory *?theory-name* for the ongoing demonstration.

(WFF ?formula) [Predicate]
This is a second order predicate that allows you to prove the truth value of the well formed formula *?formula*

If you load only the LOGIC files, this is the environment you have. On Xerox machines, you can also load the file LOGIC-DEVEL, that allows you to have the development environment: a new entry in your background menu is created, and so you are able to open a logic demonstration window.
This is the control menu:



SHOW-PROFILE: shows the current profile: the MODE of demonstration (FIRST, ALL, INTERACTIVE), and the tracing flags on unifier and solver

TRUTH VALUE ONLY: this flag controls if the prover returns all the goals with the variables instantiated or only the values T or NIL

SHOW AXIOM: shows the definition of an axiom or of a semantic attachment

EDIT AXIOM: edits the definition of an axiom or of a semantic attachment

DELETE AXIOM: deletes the definition of an axiom or of a semantic attachment

SET MODE: sets the mode of the demonstration: this may be ALL, FIRST or INTERACTIVE

TRACE SOLVER: the solver is the procedure of the interpreter that takes as arguments a tree, a formula and the clauses for that formula, and gives back the new tree obtained by the resolution operation; its behaviour is traced on a debugging window which has the middle menu capability of dribbling; the output file has the extension TRC.

TRACE UNIFIER: traces the going on of the unifier on a debugger window; this window too has the middle menu capability of dribbling its output. The pattern, the datum and the unification environment will be shown to the user.

CREATE THEORY: creates a new theory

DELETE THEORY: all the theories loaded are showed in a tablebrowser at the left of the main window; when you select one or more theories, this means that you want to use them for your demonstration; this command deletes the selected theories; you can however undelete or expunge them with the subitems of this entry. Remember that, for undeleting the selected theories with the tablebrowser mark( ), you must click middle button on it and press CTRL (PROP) key

MERGE THEORIES: merges the selected theories in a new theory; the user is prompted for the name of the new theory

LOAD THEORY: loads a theory from a file in the current directory

SAVE THEORY: save the selected theory(ies) on the corresponding files

ERASE ENV: erases all the environment of the window

EXIT: exits from the environment

Remember that, for every demonstration requested, there must be at least one theory selected in the tablebrowser at the left of the main window

I hope these notes help you to use LOGIC.

You can find some examples in the theory file LOGIC- EXAMPLES.LGC.

Any suggestion is welcome: since it is not fully tested, please notify every kind of error or bug you will find.

EXAMPLES

Choose LOGIC from the background menu: a new window will appear: choose LOAD THEORY from the control menu and type in LOGIC-EXAMPLES at the request in the prompt window: mark the theory loaded in the theories window and try:

((APPEND ?A ?B (1 2 3)))

the system will respond you

((APPEND NIL (1 2 3) (1 2 3)))

Click now on SHOW PROFILE: you will see

MODE: FIRST /Unifier: NOTRACE /Solver: NOTRACE /Values: NIL

Choose SET MODE ALL (submenu) and retry the same goal as before: you get the answer:

((APPEND NIL (1 2 3) (1 2 3)))

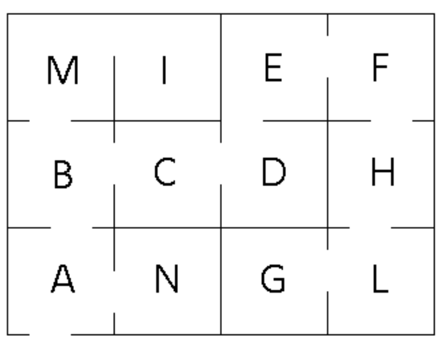
((APPEND (1) (2 3) (1 2 3)))

((APPEND (1 2) (3) (1 2 3)))

((APPEND (1 2 3) NIL (1 2 3)))

NIL

In the theory LOGIC- EXAMPLESa simple little maze is described: type in the goal:
((search a g))
that will find a path from the room ’a’ to the room ’g’.



Here are other examples of goals you can try:
((sa-member 3 (1 2 3 4 5)))
((logic-member 3 (1 2 3 4 5)))
The first one is a SA, the latter is a predicate.
((NOT (A 1))) --> T
((NOTMEMBER 2 (1 3 4))) --> T
and so on.
Try now all the other features of the language.
You can ask the system for the same goals from the lisp listener:
(load-theory ’logic-examples)
(prove ’((APPEND ?X ?Y (1 2 3 4)) ’(logic-examples)) --> T

```
(all '(?X ?Y)  '((APPEND ?X ?Y (1 2 3 4)) '(logic-examples))
--> ((NIL (1 2 3 4)) ((1) (2 3 4)) ((1 2) (3 4)) ((1 2 3) (4)) ((1 2 3 4) NIL))
```

Have fun!

LOGTIME

Johannes A. G. M. Koomen
Koomen.wbst@Xerox
Koomen@CS.Rochester.edu

October 12, 1989

SUMMARY

LOGTIME is a facility for keeping track of how you spend your time. Clicking on the background menu entry "Log Time" loads a data file and starts up a process. The process wakes at regular intervals and prompts the user for the activity engaged in during the latest interval. The default response is the previous activity. Clicking the right mouse button on the prompt window brings up a menu of known activities. Control-E in the prompt window causes the latest interval to be ignored. Clicking on the background menu entry "Log Time" when a LOGTIME process is already active causes the process to be awoken immediately. Rollout menu entries provide a way to edit the current data, to report on the data accumulated for the current day, and to stop keeping track of time (either "Quit" which updates the datafile, or "Abort" which throws away current data).

DESCRIPTION

(LOGTIME.START *datafile*) [Function]

Starts and returns a new LOGTIME process to keep track of time, using *datafile* as the file to load and store data, unless a LogTime process is already running, in which case NIL is returned. If *datafile* is NIL, it defaults to LOGTIME.DATAFILE (see below).

(LOGTIME.UPDATE) [Function]

Wakes up the LOGTIME process, if any, causing an immediate prompt for an activity.

(LOGTIME.STOP *abortflg*) [Function]

Stops and returns the LOGTIME process currently running, if any. If *abortflg* is NIL, the datafile indicated in the preceeding call to LOGTIME.START is updated.
The "Quit" roll-out entry on the background menu invokes (LOGTIME.STOP).
The "Abort" roll-out entry on the background menu invokes (LOGTIME.STOP T).

(LOGTIME.EDIT *olddatafile newdatafile*) [Function]

Loads *olddatafile* if necessary. Invokes the Lisp editor on a list of entries, where each entry is a list of starting date, ending date and activity, e.g. ("10-Oct-89 13:07" "10-Oct-89 13:12" "Mail").
If all entries can be reparsed upon return from the editor the dataset is modified accordingly. If the data did not come from the LOGTIME process currently running (if any) the data are written back to *newdatafile* if given, otherwise to *olddatafile*.
The "Edit" entry on the background menu invokes (LOGTIME.EDIT).

(LOGTIME.REPORT *bydateflg verboseflg fromdate todate datafile reportfile*) [Function]

Generates a report of the data in *datafile* (defaults to LOGTIME.DATAFILE). If *bydateflg* is non-NIL the report is sorted by date, otherwise by activity. If *verboseflg* is non-NIL, the report lists each single entry in the dataset, otherwise only cumulative time (per date or per activity) is given. If *fromdate* is given (a string acceptable to IDATE) all entries prior to this date are skipped. If the time is omitted from *fromdate* it is assumed to be 0:00. If *todate* is given (a string acceptable to IDATE) all entries past this date are skipped. If the time is omitted from *todate* it is assumed to be 23:59. The report will be generated on *reportfile* (which defaults to LOGTIME.REPORTFILE).
The "Report" entry on the background menu invokes (LOGTIME.REPORT T T <today>).

LOGTIME.INTERVAL [Variable]

The number of minutes the LOGTIME process will sleep in between prompts. Initial value is 15.

LOGTIME.DATAFILE [Variable]

The name of the default datafile. Initial value is (CONCAT (DIRECTORYNAME) "LogTime.Data").

LOGTIME.REPORTFILE [Variable]

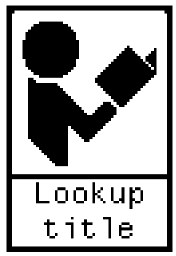
The name of the default reportfile. Initial value is T.

LOGTIME.PROMPT.URGENCY [Variable]

The value used for the PROMPTFORWORD argument *urgency.option*. Initial value is 'TTY.

LOOKUPINFILES

By: dgb (Bobrow.pa@Xerox.com)



INTRODUCTION

The LOOKUPINFILES package is a facility for building quick and easy access to on–line files. It allows search for a target string though all files in a specified list. It finds the target, and brings up the file in a window, with the target selected in inverse video. The file can then be used as the source for text for other documents. It is the basis for the user facilities of ADDRESSBOOK and FIND-CITATION. Its interface is defined by the function:

(MakeLookupWindow fileList processName mainWindowRegion iconBM iconMask iconPosition iconTitle)

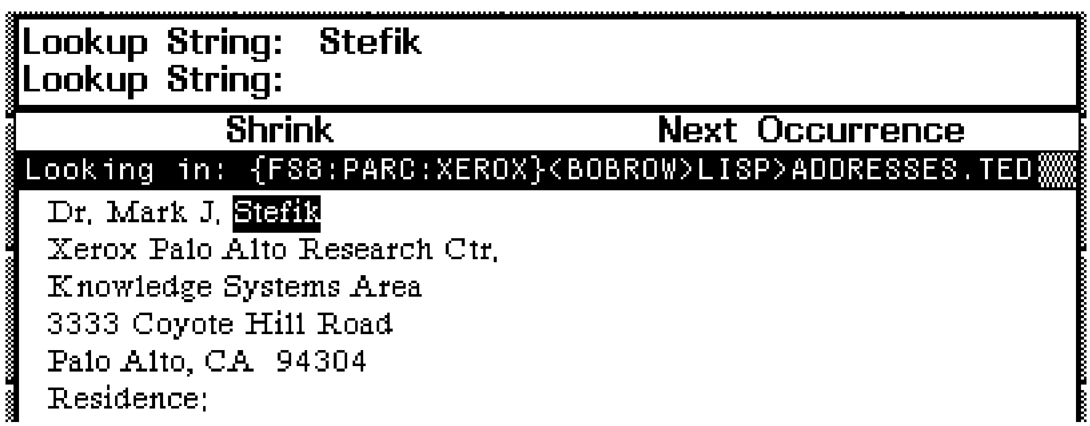
These arguments are used as follows:

fileList	List of file names. Search goes through these files in order
processName	Name appearing in PSW for this lookup process
mainWindowRegion	Region for window showing text found
iconBM	Bit map for icon when mainWindow is shrunk
iconMask	Mask for icon
iconPosition	Position for icon
iconTitle	Title put under icon to distinguish lookup operation

Arguments other than fileList are optional. Calling MakeLookupWindow will construct a Lookup window, and shrinks it to the icon provided. Opening this icon shows the window interface to the search process. To find any string in one of the files, type the string followed by a return. The program will quickly search through the files and show you an occurrence of the string typed. The located string is shown in inverse video. The title of the window will contain the name of the file in which the entry was found. The search ignores case; e.g. "bobrow" matches "Bobrow". The text of the document is scrollable, and any portion can be shift selected into another document.

Type carriage return, ^X, or click on **Next Occurrence** to search further in the files for the same string. If no (further) occurrences are found, the text window will display a message indicating the failure. Searching again after failure will start the search from the beginning of all the files, using the same lookup string. Typing a new string can be repeated as many times as you like. When you are done, just SHRINK the window back to its icon by using the Shrink selection in the title bar .

The window below is taken from the use of this package as an online address book.



Example LOOKUPINFILES window

Notes

Caching Files

When you first create the window, the program will copy the files to {CORE}, significantly speeding up queries. Bugging in the title of the main window with the left or middle mouse button will produce a menu with an option to recache all these files.

Editing Your Files

To edit the file in which a string is found, click middle button in the title of the main window, and select the option "Edit file named in window title". A TEDIT process editing the file will be set up. This process is independent of the lookup process. To select the file to be edited, rolloff the above item, and select "Select file to edit". A menu of files used by the Lookup process will be presented to you. Selecting one will cause that file to be edited.

To make editing changes visible to the lookup process, PUT the file in TEDIT; when it is done, recache the the file in core. To recache just the file edited, (the one specified in the title bar of the window), select the option "Recache file named in window title" in the middle button title bar menu. You can recache all files by selecting the option "Recache all files" in the title menu (a subselection of the item "Recache file named in window title".

Adding to the List of Files

To add to the list of files being used for lookup, select the option "Add new file" in the title bar menu. This file will be added to the beginning of the list of files to be searched, and cached in core.

Deleting a file from the List of Files

To delete from the list of files being used for lookup, select the option "Delete file from list" in the title bar menu. This file will be deleted from the list of files to be searched.

MACWINDOWS

Changes shrinking and expanding icons with a zoom.

MAGNIFIER WINDOW

By: Richard R. Burton (Burton.PA @ Xerox.Com)

This document last edited on March 17, 1986.

INTRODUCTION

File: MAGNIFIER.LCOM

Tired of giving demos in which only the two people sitting next to you can see the screen? This small package implements magnifying windows, windows that show an enlarged copy of that portion of the screen that is around the cursor. A magnifying window can be created either by calling the function MAGNIFYW or by selecting the item "Magnifier" from the background menu. A magnifying window can be made to any size and is distinguished by its large border. Once a magnifying window has been created, it can be activated by clicking the left button in it. While activated, the cursor will be replaced by a black rectangle and the contents of the rectangle will be displayed in the magnifying window enlarged by a factor of 4. The contents will continue to track the location of the cursor until the left button is clicked a second time. The magnifier can be reshaped.

Suggested use: When six people drop into your office unannounced for a demo, create a magnifying window across the top or bottom of your screen (so the people in the back can see it easily). When it is important for people to read what you are talking about, move the cursor into the magnifier, click the left button, move the cursor over the area of interest and, when the image in the magnifier has what you want, click the left button again. This will leave an enlarged part of the screen in the magnifier and free the mouse of other things. You can leave magnifier active but it will not block (so no other processes get to run) and if you move the cursor, the image in the magnifier will move too.

MakeGraph

By: D. Austin Henderson, Jr.

Supported by: Nick Briggs (Briggs.pa@Xerox.com)

INTRODUCTION

MakeGraph is a module which sits on top of Grapher and helps one create graphs depicting a data structure by walking through it. The central idea is that each point in the walk (and node in the graph) is characterized by a datum/state pair and motion is defined by a graph specification in the form of state transition function. This function is specified by a collection of state specifications, each of which indicates how to display (label and font) the datum when one is in that state and how to find the datum/state pairs which are the sons of that node. Also the state specification may specify additional roots for the walk. The generation of a branch of the graph ceases when either there are no sons of a node, or an already encountered node is revisited (identical datum and identical state). The module contains a function for creating such graphs and an example of its use: a function which graphs the graph specifications themselves. Comments are welcomed

FUNCTIONS

The main functions are:

(MAKE.GRAPH *WINDOW TITLE GRAPH.SPECIFICATION ROOTS CONTEXT LEFTBUTTONFN MIDDLEBUTTONFN TOPJUSTIFYFLG DEPTH*) [Function]

Creates a MAKEGRAPH window. If *WINDOW* is NIL, then a new one will be created and the user will be prompted to position it. Otherwise, the graph will be shown in *WINDOW*. The window will be titled with *TITLE*, will call *LEFTBUTTONFN* and *MIDDLEBUTTONFN* on nodes selected (or NIL if selection is made where no node is positioned), and will be justified as indicated by *TOPJUSTIFYFLG* (a la Grapher). The button functions are defaulted to MAKE.GRAPH.LEFTBUTTONFN (which scrolls the window so that the selected node is in the middle of the window, or if the left shift key is depressed, prints out information about it) and MAKE.GRAPH.MIDDLEBUTTONFN (which provides a menu of two choices: INSPECT - inspects the datum of the node selected, or if the left shift key is depressed, inspects the node itself; and SUB.GRAPH - which opens another MAKEGRAPH window with the same parameters as this one, but with graph starting at the selected node). The arguments to MAKE.GRAPH are added as properties to the window under their argument names. Selecting in the title invokes the functions which are the values of the window properties TITLE.LEFTBUTTONFN and TITLE.MIDDLEBUTTONFN (not in the calling sequence; set by the user if desired; called with a single argument - *WINDOW*; defaulted to a function which provides a menu of UPDATE and SHOW.GRAPH.SPEC (see functionality below)). The graph is created according to the *GRAPH.SPECIFICATION* (see below) to depth *DEPTH*, starting from *ROOTS* which are (DATUM . STATE) pairs. *CONTEXT* is an extra argument which is a passed along to all accessing expressions.

GRAPH.SPECIFICATION [Parameter]

A GRAPH.SPECIFICATION is a property list of STATE.SPECIFICATIONs where the properties are the state names.

STATE.SPECIFICAITON [Property list]

A STATE.SPECIFICATION is a property list whose properties and values are as follows (in this, EXPR means a LISP form which will be evaluated in an environment in which DATUM is bound to the node's datum, STATE to the node's state, and CONTEXT to context):

LABEL [Property]

an expression returning something which will be printed as the label of the node; if no LABEL property is provided, the string "???" will be used.

FONT [Property]

an expression returning the font to be used for this node; if no FONT property is provided, the default font for the grapher will be used.

SONS [Property]

a form indicating a list of (DATUM . STATE) pairs to be used in generating the sons of this node; the acceptable forms are any of the following:

(data-expression state-expression) [Property value]

where data-expression returns a list of datum's for the son nodes, and state-expression is evaluated in the context of each of these in turn to produce the corresponding state of each.

(LIST (data-expression state-expression) ...) [Property value]

a template of expressions which are evaluated individually to produce a list of sons of the same form, viz. (DATUM . STATE) pairs.

(EVAL expression) [Property value]

the expression returns a list of (DATUM . STATE) pairs of the sons.

(UNION sons-spec ...) [Property value]

where each sons-spec is any of these forms (recursively).

(TRACE sons-spec) [Property value]

a device for helping debug graph specifications; the value is the value of sons-specs; the user is given the chance to INSPECT them after they have been generated.

ROOTS [Property]

like SONS, except the resulting (DATUM . STATE) pairs are used as possibly additional roots of the graph.

(MAKE.GRAPH.CONSTRUCT *GRAPH.SPECIFICATION INITIAL.ROOTS CONTEXT DEPTH*) [Function]

This is the functional heart of MAKE.GRAPH broken out for those who wish to handle their own interactions with grapher and the window package. It produces a list of graphnodes with labels and sons as specified by *GRAPH.SPECIFICATION* (see MAKEGRAPH), starting from *INITIAL.ROOTS* which are (DATUM . STATE) pairs. *CONTEXT* is an extra argument which is a passed along to all accessing expressions. Returns the list of graphnodes.

(MAKE.GRAPH.FIND.ROOTS *GRAPH.SPECIFICATION INITIAL.ROOTS CONTEXT DEPTH*) [Function]

Finds the real roots from a set of initial roots, using the same processing as MAKEGRAPH uses. This is helpful when you want to hand a "correct" set of roots of a structure to MAKEGRAPH without having to explore the dependencies within that structure. As with MAKEGRAPH, the data structure is processed according to the *GRAPH.SPECIFICATION* (see MAKEGRAPH), starting from *INITIAL.ROOTS* which are (DATUM . STATE) pairs. *CONTEXT* is an extra argument which is a passed along to all accessing expressions. Returns the real roots as a list of (DATUM . STATE) pairs.

Supporting Functions

(MAKE.GRAPH.UPDATE.WINDOW *WINDOW*) [Function]

Uses the window properties (which may have been changed) to reinvoke MAKE.GRAPH on the window.

(MAKE.GRAPH.SHOW.SPEC *GRAPH.SPECIFICATION*) [Function]

Uses MAKE.GRAPH to produced a graph of a *GRAPH.SPECIFICATION*. It uses as the graph.specification for this layout the value of the variable MAKE.GRAPH.SPEC.SPEC which presents *GRAPH.SPECIFICATION* (reflectively) as a graph.specification. (MAKE.GRAPH.SPEC.SPEC can serve as a template for other graph.specifications. It is a fairly complex 9-state specification. For a simpler example see below under MAKE.GRAPH.SHOW.LIST.)

(MAKE.GRAPH.EXAMPLE.1) [Function]

Calls MAKE.GRAPH.SHOW.SPEC on MAKE.GRAPH.SPEC.SPEC.

(MAKE.GRAPH.SHOW.LIST *OBJECT*) [Function]

Uses MAKE.GRAPH to produced a graph of an arbitrary Lisp object. It uses as the graph.specification for this layout the value of the variable MAKE.GRAPH.LIST.SPEC which presents *OBJECT* as a tree whose nodes are LISTPs and whose leaves are non-LISTPs.

MAKE.GRAPH.LIST.SPEC [Variable]

MAKE.GRAPH.LIST.SPEC is the simple 1-state specification below, included here as an example of a graph.specification

```
(OBJECT (  DOC (ANY LISP OBJECT)             - some documentation
           LABEL (COND           ((LISTP DATUM) "( )")
                      (T DATUM))
           SONS ((COND           ((LISTP DATUM) DATUM)
                      (T NIL))
               (QUOTE OBJECT))
        )
 )
```

For a more complex example see above under MAKE.GRAPH.SHOW.SPEC.

(MAKE.GRAPH.EXAMPLE.2) [Function]

Calls MAKE.GRAPH.SHOW.LIST on MAKE.GRAPH.LIST.SPEC; that is, produces a graph of this simple graph.specification as a list. Notice that selecting the title command UPDATE in this window will yield a different graph of the same structure, viz. as a GRAPH.SPECIFICATION.

Other useful functions

(MAKE.GRAPH.DATUM *NODE*) [Function]

Returns the DATUM associated with the graph node *NODE*.

(MAKE.GRAPH.STATE *NODE*) [Function]

Returns the STATE associated with the graph node *NODE*.

(MAKE.GRAPH.FATHER *NODE*) [Function]

Returns the graph node which is the father of the graph node *NODE*.

MANAGER

By: Jay Ferguson, Larry Masinter and Andrew Cameron III

Maintained by Ron Fischer (Fischer.pa@Xerox.com)

Uses: MASTERSCOPE

INTRODUCTION

In its latest incarnation Manager supports MasterScope and improves its performance.

USING MANAGER:

Manager provides a way to perform most common File Manager operations onscreen using menus, both pop-up and permanent. Activity centers around the filelst, or main, menu, and menus of items of a type in the file (like all FNS, or all VARS).

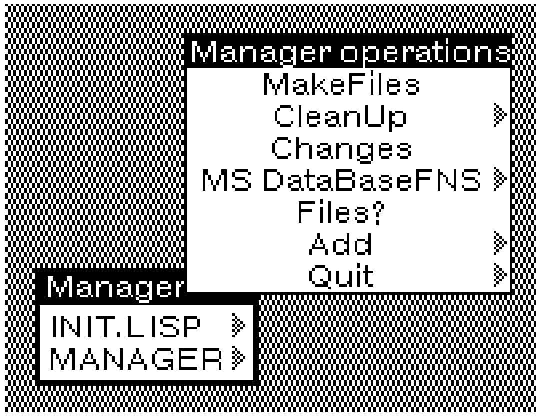
Printing and interaction is done through the Manager Command Activity Window. The first time it is needed you'll be prompted to size it onto the display. Thereafter, it will be used as needed. If shrunken before use it will wait 10 seconds after an operation and then shrink down again.

The FILELST menu

The manager provides a menu of the FILELST:

The names in the FILELST menu can be copy selected.

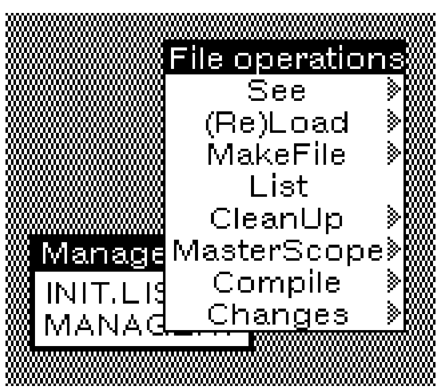
Middle buttoning on the title bar of the FILELST menu pops up a menu of operations which are applied to all loaded files:



These operations are the same as the similarly named functions in the File Manager interface, except for the following slide off options:

CleanUp:
    Set default: TCOMPL, the default compiler will be TCOMPL.
    Set default: CL:COMPILE-FILE, the default compiler will be CL:COMPILE-FILE.
MS DataBase FNS:
    various MasterScope database flags can be set
Add, notice a file via:
    LOADFNS
    LOADFROM
    LOAD
    ADDFILE*
    Edit FILELST, edit the FILELST directly in a lisp editor window.
Quit:
    Quit*, shut down the Manager, all menu caches cleared, windows closed.
    Reset, shut down and turn on the Manager again.

Left buttoning on a file in the FILELST menu (without sliding off) pops up a menu of operations on that file:



See:
- fast*, prints the source of the file.
- scrollable, displayed in a scrollable TEdit window.

(Re)Load:
- Load*, use current DFNFLG settings.
- Sysload, load with File Manager turned off.

MakeFile, dump the file
- MakeFile*, dump the source of the file by remaking it.
- New, dump the source without copying unchanged defs from existing file.
- Fast, dump source without prettyprinting (fast).
- CommonLisp, dump source in commonlisp format (loads common-makefile if needed).

List, list the source file on the default printer.

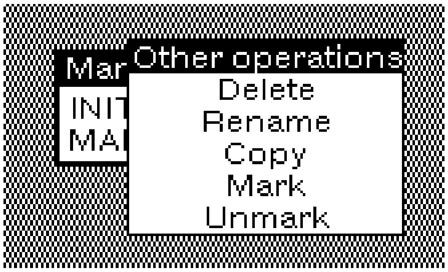
CleanUp:
- CleanUp*, dump the file according to CLEANUPOPTIONS.
- Set default compiler: TCOMPL.
- Set default compiler: CL:COMPILE-FILE.

MasterScope:
- Analyze*, analyze the fns on the file.
- Check, check the file for problems.
- Show Paths, show paths of function calls on this file.
- DatabaseFNS, display the database property for this file (loads databasefns if needed):
  - Set to ASK, ask about saving MS DB information.
  - Set to ON, automatically maintain MS DB information.
  - Set to OFF, do not save MS DB information.
  - Load DB, load an existing MS DB for this file.
  - Dump DB, dump the current MS DB for this file.

Compile:
- Compile*, compile the file based on the current settings.
- CL:COMPILE-FILE, compile the file with CL:COMPILE-FILE.

Changes:
- Brief*, prints the changes that have been made to this file.
- Everything, prints the complete list of files changes.
- Edit PL, brings up a lisp editor on the file's property list.

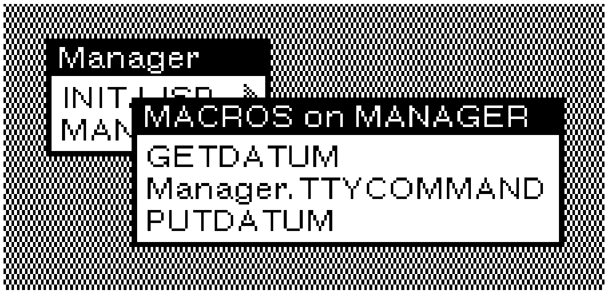
Middle buttoning on a file in the filelst menu (without sliding off) pops up a menu of generic operations on that file:



Delete, removes the file object from the system.
Rename, prompts for a new name and renames the file.
Copy, prompts for a new name and copies the file under that name.
Mark, mark the contents of the file as changed.
Unmark, unmark the contents of the file as changed.

Left buttoning on a file and sliding off to the right pops up a menu of types in the file:

Releasing on one of these places a menu of items of that type on the file:



This menu is not pop-up and remains on the display.

The items of a type menu:

These menus contain the names of all instances of a particular type on a file. Names of items in these menus can be copy selected.

Left buttoning on an item name pops up a menu of operations on that type:

Edit, brings up the source text of the item in a lisp editor.
PrettyPrint:

- Show*, prints the source text of the item quickly.
- Value, prints the global value of the item's name (assumed a symbol).
- Function Def, prints the global function definition of the item's name (assumed a symbol).
- Property List, prints the global property list of the item's name (assumed a symbol).

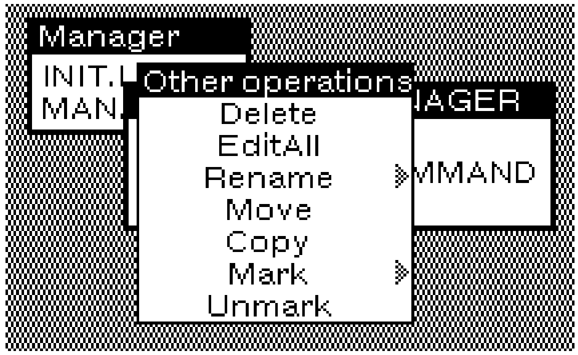
Documentation:

- Documentation*, prints the item's documentation string.
- Describe, calls describe on the item's name (assumed a symbol).

The menu of item operations shown above is the general one. There are special menus for the following types:

FNS, FUNCTIONS, RECORDS, VARS

Middle buttoning on an item name pops up a menu of generic operations on that type:



Delete, removes this item from its file.
EditAll, edits all occurances of this item's name in the latest source file (uses EDITCALLERS).
Rename:

Rename*, rename this item in its file and update all uses of the name.
CopyDef, copy this item under a new name.
Rename All, rename this item in *ALL* loaded files.

Move, move this item into another file.
Copy, copy this item into another file.
Mark:

Changed*, mark this item as changed by being edited.
Defined, mark this item as changed by being defined.
Deleted, mark this item as changed by being deleted.

Unmark, unmarks the source of this item as being changed (marks it "unchanged).

The file's makefile-environment has its readtable argument used to bring up thelisp structure editor properly on objects in the file. When SEdit is the lisp editor, the package used depends on SEdit's "correct package" heuristic (usually that of the symbol naming what is being edited).

Loading and controlling Manager:

Just load the file. Manager can be started either from the background menu or by calling the FNS MANAGER (see below).

Programmer's interface to Manager:

(MANAGER *POSITION*) [FNS]

Starts up the manager. If *POSITION* is given, the filelst menu will be appear there.

(MANAGER.RESET *RESTARTFLG*) [FNS]

Shuts down the manager. If *RESTARTFLG* is true, manager will be immediately restarted after the shutdown.

Manager.SORTFILELSTFLG [INITVAR]

If true, the FILELST will be sorted, without side effecting the actual FILELST variable. If unset, defaults to T.

Manager.MENUROWS [INITVAR]

Maximum number of rows in a manager menu. If unset, defaults to 20.

manager-marked-shade [INITVAR]

The shade used to indicate that an item has been marked as changed. If unset, defaults to MENUBOLDFONT.

Change History

This is a history of edits made to the Manager. Please add your initials and a short description of what you changed to the END of the file. Be sure to include the name of the definition you modified.

andyiii- All menus are sorted now.

andyiii - Appropiate sub-menu update when something is changed that they contain.

andyiii- un-marking a file in the main menu now works and updates all the sub-menus of that file.

andyiii - added option to MAKEFILE menu item for files to write CommonLisp source using common-makefile.

andyiii - added commonlisp DESCRIBE for items

andyiii - Added a way to add files to the file managers main menu

andyiii - Can edit files property list from CHANGES menu

andyiii - Can now mark a whole file from main menu

andyiii - Can chose between TCOMPL (.LCOM files) and compile-file (.dfasl files) This is awkard since is uses the global variable *default-cleanup-compiler*

andyiii - Can get CommonLisp documentation string and descriptons

andyiii - Can now PrettyPrint a value, function def, or prop list and also show how the item would be written to a file

andyiii - Cleaned up specialized menus for FNS, FUNCTIONS, VARS and PROPS

andyiii - All dialog now goes through the MANAGER ACTIVITY WINDOW

RAF 7/31/87 - Fixed the rename option to not specify a source file, uses the ? search (core then file).

RAF 7/31/87 - Added an "edit all occurances of item’s name" option to file relations menu.

RAF 7/31/87 - Manager.ACTIVEFLG is now a special that is bound by all advice to avoid redundant updates inside of themselves. This is a big speed improvement!

RAF 7/31/87 - Fixed Manager.HASITEM and Manager.HIGHLIGHT to use SASSOC, so that list items in menus get highlighted properly.

RAF 7/31/87 - Middle button on Manager file menu now brings up rename, etc. Used to bring up coms to edit (inconsistent).

RAF 7/31/87 - Main menu flashes if bad button/command is given.

RAF 8/4/87 - MANAGER-ADDTOFILES? now initialized to NIL, reducing redundant updates.

RAF 8/14/87 - In Manager.ALTERMARKING: removed extra code which tracked the files containing updated menus. Removed call to Manager.CHECKFILE. Made call to Manager.MAINUPDATE pass T if the reason for marking was DEFINED or DELETED; these cases also call Manager.COMSOPEN.

RAF 8/15/87 - In Manager.DO.COMMAND: moved binding of ACTIVITY-WINDOW-WAS-SHRUNK into the form eval'ed in the process where references are made. Moved setting of ACTIVITY-WINDOW-WAS-SHRUNK after the spot where its referent ACTIVITY-WINDOW is initialized.

RAF 8/16/87 - Advice for LOAD and LOADFNS now call Manager.CHECKFILE instead of Manager.MAINUPDATE (latter only does highlight updating, former can rebuild main menu). Advice for ADDTOFILES? now doesn't disable manager inside of its advised form, so that the ADDTOCOMS and DELFROMCOMS advice will work.

RAF 8/17/87 - Added Manager.FILELSTCHANGED? (which is tricky, since sorting in the main menu changes its order). Manager.CHECKFILE now tests whether the file being checked is in the main menu. If not the main menu is rebuilt. MANAGER fns disables manager around its call to UPDATEFILES. Manager.GETFILE takes a prompt argument (which is now passed in by Manager.DO.COMMAND).

RAF 8/18/87 - Manager.REMOVE.DUPLICATE.ADVICE now disables the manager when it manipulates the advice (to avoid animating the changes in the menus). The advice on LOAD and LOADFNS now call Manager.REMOVE.DUPLICATE.ADVICE.

RAF 8/20/87 - Fixed Manager.MAKEFILE.ADV to handle atomic cleanup options. Also made the top level Manager.RESET call take Manager.ACTIVEFLG, so that manager stays on when reloaded if it was on already. Manager.REMOVE.DUPLICATE.ADVICE now removes *all* duplicates of the first piece of advice (rather than only the second).

RAF 8/21/87 - Made MANAGER-WINDOWS be an initvar so that Manager.RESET from top level sees the right thing on first startup.

RAF 9/2/87 - Changed the manager shrunken bitmap to something more respectable. Added ADVISE and UNADVISE menu options for the ADVICE definer. Added a "Show all advice in effect" option to the manager main window middle button menu. Changed the messages printed out by Manager.DO.COMMAND to all use printout and lambdafont for highlighting.

RAF 9/3/87 - Added a clause in the startup fns MANAGER which reports when FILELST is empty and manager can't start. Also fixed a bug in where marking a file didn't bold the main menu entry (added an updatefiles in Manager.ALTERMARKING). Also caused the advice on the "redundant" call to (MARKASCHANGED :IN DEFAULT.EDITDEFA0001) to fire when FILELST is being edited (seems it was the only way to call markaschanged in that one case).

RAF 11/18/87 - Changed the call to EDITDEF in Manager.DO.COMMAND to include a :DONTWAIT option. The tracks a change in SEdit for the Mototwn release.

RAF 11/18/87 - Added some type checking to the sort testing function Manager.SORT.COMS so that it doesn't convert its arguments to strings unless they're not LITATOMS. This should make menu generation alot faster.

MATHTONS

By: Tad Hogg (Hogg.pa@Xerox.com)

INTRODUCTION

This file defines the translation array needed to convert from the Press MATH font to corresponding NS characters. This allows documents containing the MATH font to be printed on Interpress printers.

The array \MATHTONSARRAY contains the translations for most of the characters. Some may not be available on particular printers, causing them to appear as black boxes.

Contains a tool for translating File Manger format Interlisp source files from Medley into Common Lisp text files. The software runs in the Medley system.

MISSILE

By: Anonymous
Maintained by: Frank Shih (Shih.envos@Xerox.com)

INTRODUCTION

MISSILE is a Lisp version of the video game, Missile Command. It was discovered on a file server after a Lisp class.

STARTING MISSILE

Load MISSILE.LCOM, and then call (IL:INIT-MISSILE). Try to destroy incoming missiles by clicking the mouse button in the sky. The rest should be obvious. Warning: the game makes lots of noise.

RESTARTING MISSILE

After a game of MISSILE, it can be restarted from the background menu.

MODERNIZE

By Ron Kaplan

This document was last edited in May 2022

MODERNIZE is a simple Lispusers package that changes the mouse actions on Medley windows so that moving and shaping can be done in a way that approximates the behavior of windows on modern platforms, Mac, Windows, etc. It also adds some meta keys to emulate more conventional behavior.

Thus, for a window that has been created or transformed in this way, you can move the window by left-clicking in the title bar and dragging the window's ghost region. Or you can reshape by clicking in a corner of the title bar or near the bottom of the window to drag out the ghost region by that corner.

The menu behavior for other buttons or buttons clicked in other positions is unchanged.

For bottom corners, "near" means inside the window within `MODERN-WINDOW-MARGIN` (initially 25) pixels above or to the left/right of the corner.

For top corners, "near" means within the title bar and within the margin from the left/right edges. (Windows that don't have a title-bar, like Snap windows, can be set up so that moving can happen by clicking anywhere, and shaping at the top is determined by the margin inside the window region.

The function `MODERNWINDOW.SETUP` establishes the new behavior for classes of windows:

```
(MODERNWINDOW.SETUP ORIGFN MODERNWINDOWFN ANYWHERE TITLEPROPORTION)
```

ORIGFN is either the name of the BUTTONEVENTFN for a class of windows (e.g. `\TEDIT.BUTTONEVENTFN` for Tedit windows) or it is a function that creates windows of a particulate kind (e.g. SNAPW or `ADD-EXEC`).

`MODERNWINDOW.SETUP` moves the definition of `ORIGFN` to the name `(PACK* 'MODERN-ORIG-ORIGFN)`. It then provides a new definition for `ORIGFN` that does the window moving or reshaping for clicks in the triggering locations, and otherwise passes control through to the original definition.

If `ORIGNFN` is a button event function, then `MODERNWINDOWFN` should not be specified. In that case a new definition for ORIGFN is constructed to provide the desired windowing behavior.

Otherwise, if `ORIGFN` is the function that creates windows of a class (e.g. `SNAPW`), then a `MODERNWINDOWFN` should be provided to create such windows (by calling `(PACK* 'MODERN-ORIG-ORIGFN)`). The definition of `MODERNWINDOWFN` replaces the original definition of `ORIGFN`.

TITLEPROPORTION may be a number between 0 and .5. It specifies the proportion from either edge of the title bar where clicks will be interpreted as window operations. For example, if TITLEPROPORTION is .25, then a title click that is up to a quarter of the way from the left or right edge of the window will trigger the moving operation. Clicks in the middle 50% of the title bar will always pass through.

If the flag `ANYWHERE` is non-`NIL`, especially for windows without a title bar, then the moving behavior is triggered by a click anywhere in the window (except the corners).

Because this works by redefining existing functions, it is important that the `MODERNIZE` package be loaded AFTER Tedit and Sedit, if those are not already in the sysout. And it should be called to upgrade the proper functions for other window classes that might later be added.

Provided these capabilities are already loaded, the following window classes are "modernized" when `MODERNIZE` is loaded:

TEDIT
SEDIT
INSPECTOR
SNAP
DEBUGGER
EXEC
TABLEBROWSER
FILEBROWSER
FREEMENU
GRAPHER
PROMPTWINDOW
SPY
TOTOPW

If it is not known or it is inconvenient to systematically upgrade a button function or a window-creation function, the new behavior can be provided after a particular window has been created, by invoking

```
(MODERNWINDOW WINDOW ANYWHERE TITLEPROPORTION)
```

This saves the windows existing BUTTONEVENTFN as a window property PREMODERN-BUTTONEVENTFN, and installs a simple stub function in its place.

If things go awry:

```
(UNMODERN.SETUP ORIGFN
```

is provided to restore the original behavior for windows whose buttonevent function is ORIGFN.

```
(UNMODERNWINDOW WINDOW)
```

restores a modernized window to its original state.

`MODERNIZE` also augments the Tedit keyboard commands, to increase compatibility with modern interfaces:

`Meta,q` is a synonym for Quit
`Meta,a` is a synonym for Select All

Known issues:

Clicking at the bottom of an `EXEC` window running TTYIN is effective only when the input line is empty.

.

MONITOR

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: COURIERSERVE, BITMAPFNS

MONITOR is a remote screen monitor which shows a scaled down version of the entire remote screen and a small section at full size which can be moved around.

The module contains the code for the client and the server and must be loaded on both. The program supports multiple instances of the tool, even at different scale factors, and works correctly between machines with different size displays.

The lower, full screen window is mouse sensitive. Pressing the left button in the window updates the upper, closeup window to contain the portion of the remote screen indicated by the cursor. Pressing the middle button in the full screen window causes the compressed image of the remote screen in the lower window to be updated.

(MONITOR *HOST [SCALE]*) [Function]

Opens a remote screen monitor onto *HOST*, where *HOST* is any specification that COURIER.OPEN accepts. *SCALE* is optional and determines the amount of compression of the remote screen bitmap as well as the amount of area covered by the closeup.

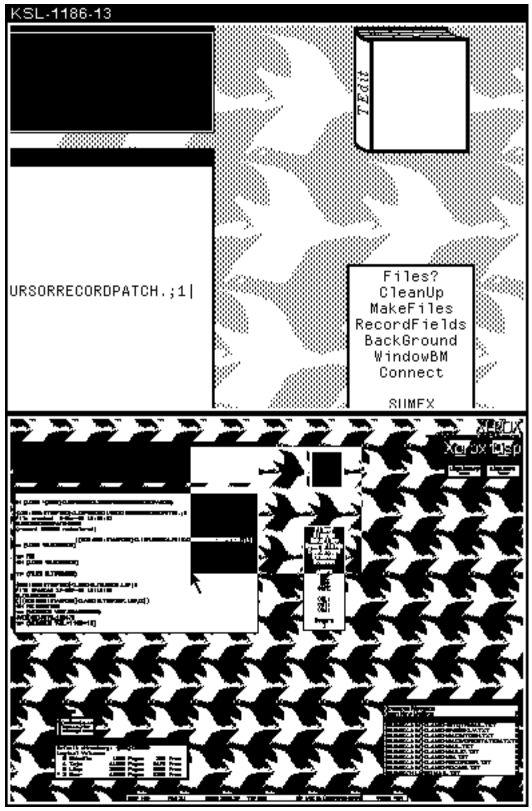
The useful range of scale factors is from **2** to about **8**; a scale factor of **N** will compress the remote screen by **1/N** in width and height and the closeup will cover $\mathbf{1/N^2}$ of the area of the remote screen.

MONITOR.SCALE = **3** [Variable]

If not specified, the *SCALE* argument to MONITOR defaults to the value of MONITOR.SCALE.

KNOWN PROBLEMS

- The Courier program number that the MONITOR Courier program uses is unregistered.
- The monitor does not yet correct for VIDEOCOLOR (which affects both the client and server).


KSL-1186-13
TEdit
URSORRECORDPATCH.;1|
Files?
CleanUp
MakeFiles
RecordFields
BackGround
WindowBM
Connect
SUMEX

NEATICONS

By: Peter Schachte (quintus!pds@Sun.com)

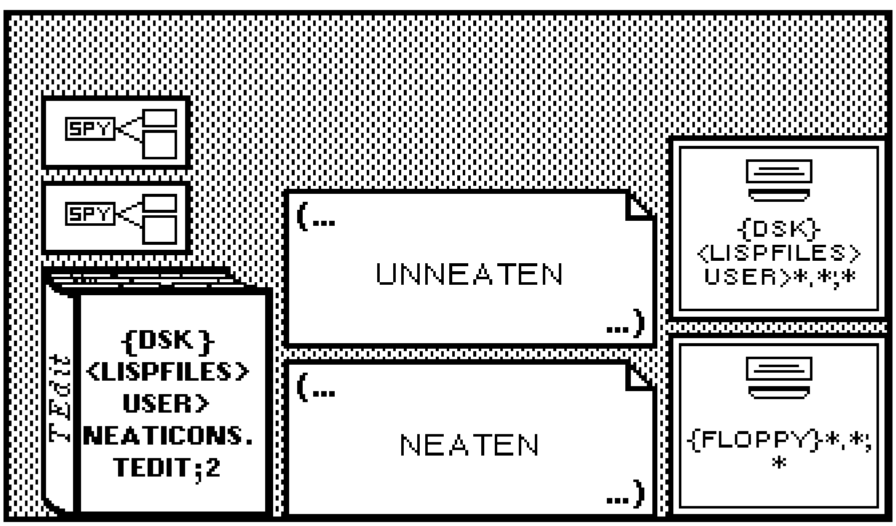
INTRODUCTION

If you like to keep your icons neatly arranged on your screen, NEATICONS is for you. After this package is loaded, whenever an icon is created by shrinking a window, that icon will be "neat." But what *is* a neat icon? A neat icon is one that is lined up with with another icon or window, or the edge of the screen. The easiest way to see this is to load the package, shrink a few windows (creating a snapshot and shrinking it is easy), and move them around. When a neat icon is moved near another icon or window or the edge of the screen, it is "grabbed" and moved neatly near it.

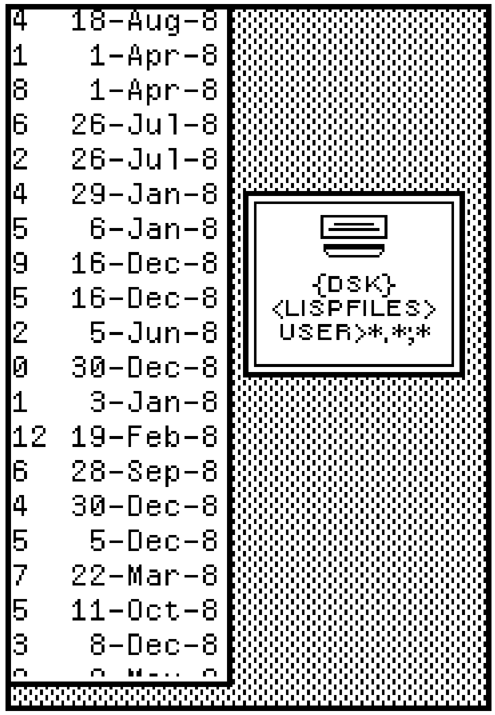
Neat icons line themselves up in a variety of ways. They will flush themselves with the edge of the screen. They will move themselves a fixed number of pixels from the edge of another window. Or they will align one of their edges with the corresponding edge of another window. When you move a neat icon, it will try to find a "neat" position near where you placed it, and place the window there instead. It may find a nearby position that is horizontally neat but not vertically, or vice versa. In any case, it will move the window into the nearest neat position it can find, or leave it where you put it if it can't find any nearby neat places.

EXAMPLES

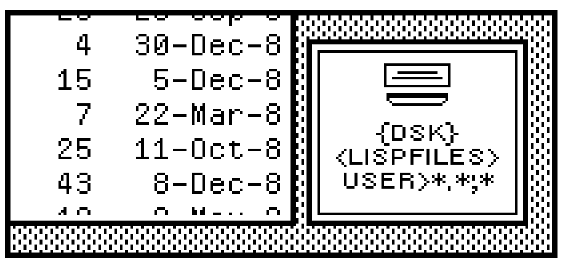
Here are a few examples of how your icons will be arranged. A typical cluster of neat icons:



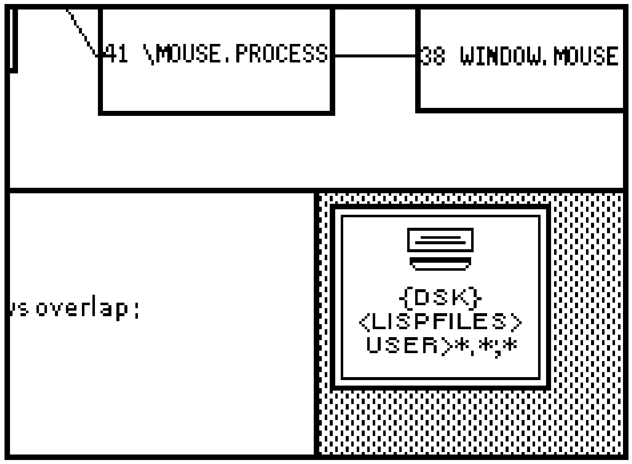
Neat icons can align themselves with the side of a window:



But more typically, they will align themselves with the corner of a window:



And they occasionally align themselves near the corner where two windows overlap:



DETAILS

You can just load this module and forget about it, and it will behave as advertised. It does have two user-settable parameters, and two user-callable functions, however. These are documented below.

Please note that the NEATICONS module is contained entirely in the NEATICONS package. This package exports the following symbols.

How near is near?

NEATICONS:DEFAULT-TOLERANCE [Variable]

This global parameter determines how many pixels vertically and horizontally an icon will be moved in order to make it neat. This defaults to 100.

Spacing between icons

NEATICONS:DEFAULT-SPACING [Variable]

This global parameter determines how many pixels apart neat icons will be placed. The default is 5.

Making a window neat

Loading the NEATICONS module causes SHRINKW to be advised, so every time a window is shrunk, the icon is made neat. So icons created before you load NEATICONS will not be neat, but they can be made neat by expanding and then re-shrinking them.

But suppose you want to make a regular window neat?

(NEATICONS:NEATEN &OPTIONAL WINDOW) [Function]

makes WINDOW neat. WINDOW defaults to (WHICHW), so you can point at a window with the mouse and type (NEATICONS:NEATEN).

Making a window sloppy

(NEATICONS:UNNEATEN &OPTIONAL WINDOW) [Function]

makes WINDOW no longer neat. It behaves just like a normal, sloppy, vanilla window. WINDOW defaults to (WHICHW).

Other MOVEFNs

The NEATICONS module uses each window's MOVEFN prop. If you wish to have another MOVEFN on a neat window, you can. If a window has MOVEFN when it is NEATICONS:NEATENed, it will be preserved. If you wish to add a MOVEFN to an existing neat window, you should put it on the window's

NEATICONS:USERMOVEFN [Window Prop]

prop. This prop should hold a list of functions. When a neat window is moved, first it finds the nearest neat place. Then the first function on the window's NEATICONS:USERMOVEFN prop is called with the neat position as argument. If this function returns IL:DON'T, the window won't be moved. If it returns NIL, the position argument it was passed is passed to the second function on the list. If it returns a position, this position is passed to the second function on the list. The result of each function on the list is treated similarly, until all the functions have been called. The latest position is used as the position to move the window to.

NOTEPAD

By: D. Austin Henderson, Jr. (AHenderson.pa@Xerox)

Compiled for Medley by Larry Masinter (Masinter.PA@Xerox.COM)

Auxiliary file: NOTEPAD-CORESTYLES

NOTEPAD is a module for creating NOTEPAD windows - windows in which one can do artwork at the bitmap level. The ideas in this module come pretty directly from other people's work, both inside and outside Xerox, including Markup, Draw and Smalltalk. Notepad as it stands is the product of about a man-week of work, using standard Interlisp-D as released and the EDITBITMAP module of bitmap manipulation functions. It provides a *nearly unusable* interface to some distinctly interesting functionality. Comments and suggestions are welcomed.

NOTEPAD (BITMAP COLORFLG)

Creates a NOTEPAD window. If BITMAP is NIL, then you are prompted for region for the window. Otherwise, a region is defined from the size of BITMAP, and you are prompted to move it to a desired position. If COLORFLG is non-NIL, the NOTEPAD window will be used only for control (for menu's, etc.), and all the painting will take place on the color screen.

There are two menus: one in the title of the window, and one in the window proper. Both are invoked by buttoning with either left or middle buttons.

Title menu

This menu gives access to commands for manipulating the window as a whole:

New Notepad

Copy Notepad

Save as a bitmap.

Window menu

This menu has two columns of commands: those in the left column of the menu are for painting/erasing material into/from the bitmap (if this menu is invoked with the left button, painting is implied; if with the middle button, erasing); those in the right column of the menu are for changing the style in which the operations work.

Painting/Erasing

You can paint/erase using trajectories, or using (or editing) single objects. The commands in the left column of the menu are divided into two sets which reflect rthis division.

Trajectories

Sketch: follows the mouse to define a trajectory of points at which to sketch.

Line: prompts for endpoints (fix and rubberband) and uses the points on the line as a trajectory.

Circle: prompts for center and a point on the circumference, and uses the points on the line as a trajectory.

Ellipse: prompts for center, end of semi-major axis and end of semi-minor axis, and uses the points on the line as a trajectory.

Open curve: prompts for first point and one or more subsequent points. You indicate that you are finished by depressing the left shift key on the last point defining the curve. A smooth curve is fitted through these points and used as a trajectory.

Closed curve: Like open curve, except that the fitted curve is closed, starting at the last point given, and proceeding to the first and then subsequent points.

Objects/editing

Text: prompts for text, and permits positioning it with the mouse.

Area of the screen: Prompts for a (rectangular) region of the screen and places permits placing them where you want in the window.

Shade rectangle: prompts for a region, and paints/erases it with the current shade (a shade of black does complete paint and erase).

Fill: Prompts for a region within which to fill, and a point within the area to be filled; fills the area (not necessarily a rectangle, but defined by being closed rectilinearly) with the current shade.

Edit area: prompts for a region of the window and invokes the (Trillium) bitmap editor (standard Interlisp-D bitmap editor is the "hand-edit" choice on the submenu; others allow reflecting, rotating, shifting, inverting, putting on borders, etc.) on it. If the bitmap resulting from the edit is the same size as the original, it replaces the original region; if not, you are promtped for a place to put it.

Style

Notepad operations (see above) are carried out in a style. The style at any given moment is given by a collection of characteristics. The current style can be saved (use the command SAVE.STYLE) under a name and restored (RESTORE.STYLE). Styles may be deleted (DELETE.STYLE) from the collection of saved styles. The style collection is currently part of notepad. Consequently, moving styles between loadups is not directly supported. (It is always possibole to save it on a separate file. The styles are stored as the value of NOTEPAD.STYLES. It includes bitmaps, and msut therefore be added as an UGLYVARS.)

The characteristics in the style are:

Brush: A bitmap which is either painted or erased at each point on or resulting from (see symmetry) a trajectory (see the operations paint, line, circle, ellipse). DEFINE.BRUSH prompts for a region which will then become the brush. EDIT.BRUSH permits editing of the brush bitmap (using the same editor as the operation EDIT.AREA - see above). BRUSH=COOKIE.CUT.WITH.MASK also defines a brush (see mask, below).

Use mask: An indication of whether of not to use the masking function (see mask, below) before painting/erasing. USE.MASK toggles this setting.

Mask: A bitmap which is used to clear out an area before the brush is used. The mask is erased if the brush is painting, and visa-versa. DEFINE.MASK prompts for a region which will then become the mask. EDIT.MASK permits editing of the mask bitmap (using the same editor as the operation EDIT.AREA - see above). MASK=OUTLINE.OF.BRUSH defines the mask to be the same size as the brush, with a pattern (of black) which is the filled-in outline of the brush. BRUSH=COOKIE.CUT.WITH.MASK allows you to move the mask over a section of the window and define the brush as the points so covered.

Use grid: An indication of whether of not to use the gridding function (see grid, below) while painting/erasing. USE.GRID toggles this setting.

Grid: An origin and the point (1, 1) of a grid to be used to "attract" the points used in following a trajectory. DEFINE.GRID prompts for the origin and (1, 1) point of the grid.

Use symmetry: An indication of what sort of symmetry function to use while painting/erasing. USE.SYMMETRY permits setting it as you choose. The choices are none, left/right, up/down, 4-fold (both left/right and up/down) and 8-fold (4-fold plus reflecting about the 45-degree diagonals). The brush/mask used when painting/erasing symmetrically can themselves be either identical to the brush/mask in use or symmetrically reflected. USE.SYMMETRIC.BRUSH/MASK toggles this setting.

Point of symmetry: The point with respect to which the symmetry functions are defined. POINT.OF.SYMMETRY prompts for this point.

Text font: The font in which text is printed. DEFINE.FONT permits choosing one of the fonts already loaded or OTHER (in which case you can type in the font description (family size face)).

Shade: The shade used for the rectangle and fill operations. EDIT.SHADE permits you to edit the shade (using the standard Interlisp-D shade editor).

NOVAFONT

By: Nick Briggs (Briggs.pa@Xerox.com)

INTERNAL

By Nick Briggs
With prodding from Larry Masinter

This utility file allows Lisp to use in fonts in the NOVAFONT format, which is used by Viewpoint.

NovaFont files have in them both the display bitmaps for all sizes of the font, and also the printer widths.

NovaFont files need to be explicitly noticed.

(notice-novafont-file filename) [Function]

After calling notice-novafont-file, the fonts in the given file name will be "known" by the environment. For example,

```
(notice-novafont-file
"{eris}<lispcore>xeroxprivate>fonts>optimamedium.novafont")
```

After this, FONTCREATE will get the bits from the file.

The novafont reader is crafted in such a manner that it only ever reads the file forwards, so you can load fonts from an NS server. For PARC users, note that there are a bunch of NovaFonts (a few malformed...!) on

{starfile public:}<vp applications>*.novafont

including all the "printwheel" fonts, all the Japanese and Chinese character sets (60Q thru 140Q approx), the PC fonts, Quartz .

Unfortunately, it currently doesn't die "gracefully" on the malformed files. It can't just call VP-FONT-P on the files and continue on because it would mean backing up the file pointer.

Instead of loading NovaFonts on demand, all of the display fonts in a NovaFont file can be loaded at once by calling

(load-novafont-file filename) [Function]

NSDISPLAYSIZES

By: Bill van Melle (vanMelle.pa@Xerox.com)

The NS font families all have screen fonts that display at approximately their nominal point size. This means that there is a closer congruence between their appearance on the display and on hardcopy than there is for, say, the Press fonts. Unfortunately, this means that the NS display fonts are "too small"—a size that is quite readable on hardcopy can be uncomfortably small on the display. The module NSDISPLAYSIZES attempts to ameliorate this problem by fooling FONTCREATE into using bigger fonts for display without changing anyone's belief in the nominal size of the font.

Loading NSDISPLAYSIZES.LCOM modifies FONTCREATE's font file lookup procedure so that a request for NS display font of size *n* actually reads the font file for size *n*+2. For example, (FONTCREATE 'MODERN 10) will actually read the display font file belonging to Modern 12, but FONTCREATE will still believe the resulting font is Modern 10. Font sizes greater than 12 are not affected, on the grounds that those fonts are already big enough to read, and not all fonts are available in size *n*+2 for large *n*; hence, for example, Classic 12 and Classic 14 will end up using the same actual font for display. Also, since Terminal 14 does not yet (as of this printing) exist, Terminal 12 remains Terminal 12.

A font is considered an NS font if its name is a member of the list NSFONTFAMILIES, whose initial value is (CLASSIC MODERN TERMINAL OPTIMA TITAN).

Loading the module clears the internal font cache of all NS display fonts, so that subsequent calls to FONTCREATE will not erroneously return a font cached earlier under the default lookup procedure. Of course, if someone has already set some font variable to (FONTCREATE 'MODERN 10), that font descriptor will not be affected.

Note that this module has no effect on hardcopy—a font is always printed at the size you named it. And you can still use TEdit's Hardcopy display mode to see how a piece of text will be formatted on the printer.

If the VIRTUALKEYBOARDS module is present, then loading NSDISPLAYSIZES automatically edits its keyboard specifications so that it continues to use Classic 12 in its keyboard displays. Without this fix, the keyboard display routines will try to create Classic 12, which NSDISPLAYSIZES coerces to Classic 14, and as a result the keyboards will be poorly displayed and lack many characters (since Classic 14 is not as complete as 12). If you load VIRTUALKEYBOARDS *after* NSDISPLAYSIZES, you should call (VKBD.FIX.FONT) yourself to make the change.

Note that other modules can have similar problems if they have hardwired into them a specific size of NS font as being appropriate for their display configuration. For such modules to operate correctly in the presence of NSDISPLAYSIZES, they might want to be aware of the function used to coerce sizes:

`(NSDISPLAYSIZE` *FAMILY SIZE FACE EXTENSION*`)` [Function]

Returns a font size that `(FONTCREATE` *FAMILY SIZE FACE*`)` will use instead of *SIZE*. *EXTENSION* must be a member of DISPLAYFONTEXTENSIONS in order that we know we are doing this for the display. Follows the rules described above. For example, `(NSDISPLAYSIZE 'MODERN 12 'MRR 'DISPLAYFONT)` returns 14. `(NSDISPLAYSIZE 'GACHA 12 'MRR 'DISPLAYFONT)` returns 12.

In the simplest case a module could just test for the existence of `NSDISPLAYSIZE` and choose one size or another. For example,

```
(SETQ MYFONT (FONTCREATE 'MODERN (if (GETD 'NSDISPLAYSIZE)
                                     then 10
                                    else 12)))
```

OBJECTWINDOW

By:

Ron Kaplan

This document edited on

December 21, 2021

An "object window" is a window that contains a sequence of arbitrary image objects arranged either vertically or horizontally. The OBJECTWINDOW package provides the functions for creating such a window, adding objects and manipulating them in various ways, and invoking their IMAGEFNS functions according to mouse or other signals.

An object window is created by the function OBJ.CREATEW:

(OBJ.CREATEW WINDOWTYPE REGION/WINDOW TITLE BORDERSIZE NOOPENFLG SEPDIST BOXFN DISPLAYFN BUTTONINFN) [Function]

The arrangement of objects is determined by the obligatory WINDOWTYPE, either VERTICAL or HORIZONTAL. The other arguments are optional. REGION/WINDOW, TITLE, BORDERSIZE, and NOOPENFLG are passed to CREATEW to create the window. If REGION/WINDOW is a window, it is converted to an object window with TITLE. Otherwise, a new window with region REGION/WINDOW if non-NIL. The objects in the window will be separated (vertically or horizontally) by SEPDIST points, defaulting to 0. The arguments BOXFN, DISPLAYFN, and BUTTONINFN are provided as default functions if is convenient to insert objects whose IMAGEFNS are not fully fleshed out. HARDCOPYFN overrides the default hardcopy function for the window, and HCPYHEADING is used instead of TITLE for hardcopy output.

(OBJWINDOWP WINDOW) [Function]

True if WINDOW is an object window.

(OBJ.ADDTOW WINDOW OBJECT) [Function]

Adds OBJECT at the end of the object sequence in WINDOW.

(OBJ.ADDMANYTOW WINDOW OBJECTS) [Function]

Equivalent to calling OBJ.ADDTOW for each object in OBJECTS.

(OBJ.INSERTOBJECTS WINDOW NEWOBJECTS OLDOBJECT WHERE) [Function]

Inserts NEWOBJECTS at position WHERE (BEFORE or AFTER) with respect to OLDOBJECT.

(OBJ.CLEARW WINDOW) [Function]

Clears the visible objects in WINDOW.

(OBJ.DELFROMW WINDOW OBJECT) [Function]

OBJECT is removed from WINDOW.

(OBJ.REPLACE WINDOW OLD.OBJECT NEW.OBJECT DONT.REDISPLAY.FLG) [Function]

Replaces OLD.OBJECT with NEW.OBJECT in WINDOW, redisplaying the visible objects unless DONT.REDISPLAY.FLG.

(OBJ.FIND.REGION WINDOW SEARCHOBJECT) [Function]

Returns the region in WINDOW occupied by SEARCHOBJECT.

(OBJ.MAP.OBJECTS WINDOW MAPFN) [Function]

Applies MAPFN to each object OBJ in WINDOW. If MAPFN returns an image object, that object replaces OBJ in WINDOW.

(OBJ.OBJECTS WINDOW) [Function]

Returns the list of objects in WINDOW.

[This file is an on-line textual version of MIT/AIM-958a. It is not as readable as the printed version of the document, but is more convenient in some ways. In order to locate the doumentation for a particular function or error message, search for two carriage returns followed by the name of the function or the number of the error. A printed copy of MIT/AIM-958a can easily be obtained from the MIT AI Lab publications office; 5454 Tech SQ Room 818; Cambridge MA 02139; (617)-253-6773.]

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 958a March 1988

Obviously Synchronizable Series Expressions: Part I: User's Manual for the OSS Macro Package

by

Richard C. Waters

Abstract

The benefits of programming in a functional style are well known. In particular, algorithms that are expressed as compositions of functions operating on series/vectors/streams of data elements are much easier to understand and modify than equivalent algorithms expressed as loops. Unfortunately, many programmers hesitate to use series expressions, because they are typically implemented very inefficiently.

A Common Lisp macro package (OSS) has been implemented which supports a restricted class of series expressions, obviously synchronizable series expressions, which can be evaluated very efficiently by automatically converting them into loops. Using this macro package, programmers can obtain the advantages of expressing computations as series expressions without incurring any run-time overhead.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

Acknowledgments.
Both the OSS macro package and this report have benefited from the assistance of a number of people. In particular, C. Rich, A. Meyer, Y. Feldman, D. Chapman, and P. Anagnostopoulos made suggestions which led to a number of very significant improvements in the clarity and power of obviously synchronizable series expressions.

1. All You Need To Know to Get Started

This first section describes everything you need to know to start using the OSS
macro package. It then presents a detailed example. Section 2 is a comprehensive reference manual. It describes the functions supported by the OSS macro package in detail. Section 3 contains the bibliography. Section 4 explains the warning and error messages that can be produced by the OSS macro package. Section 5 is both an index into Section 2 and an abbreviated description of the OSS functions. [This section is omitted in the on-line version. Use searching to find individual function descriptions.]

A companion paper [6] gives an overview of the theory underlying the OSS macro package. It explains why things are designed the way they are and compares the
OSS macro package with other systems that support operations on series. In addition, the companion paper gives a brief description of the algorithms used to implement the OSS macro package. As part of this, it describes a number of subprimitive constructs which are provided for advanced users of the OSS macro package.

The OSS data type.
A series is an ordered linear sequence of elements. Vectors, lists, and streams are examples of series data types. The advantages (with respect to conciseness, understandability, and modifiability) of expressing algorithms as compositions of functions operating on series, rather than as loops, are well known. Unfortunately, as typically implemented, series expressions are very inefficient---so inefficient, that programmers are forced to use loops whenever efficiency matters.

Obviously Synchronizable Series (OSS) is a special series data type that can be implemented extremely efficiently by automatically converting OSS expressions into loops. This allows programmers to gain the benefit of using series expressions without paying any price in efficiency.

The OSS macro package adds support for the OSS data type to Common Lisp [4]. The macro package was originally developed under version 7 of the Symbolics Lisp Machine software [7]. However, it is written in standard Common Lisp and should be able to run in any implementation of Common Lisp. (It has been tested in DEC Common Lisp and Sun Common Lisp as well as Symbolics Common Lisp.)

The basic functionality provided by the OSS macro package is similar to the functionality provided by the Common Lisp sequence functions. However, in addition to being much more efficient, the OSS macro package is more powerful than the sequence functions, because it includes almost all of the operations supported by APL [3] and by the Loop macro [2]. As a result, OSS expressions go much farther than the sequence functions towards the goal of eliminating the
need for explicit loops.

Predefined OSS functions.
The heart of the OSS macro package is a set of several dozen functions which operate on OSS series. These functions divide naturally into three classes. Enumerators produce series without consuming any. Transducers compute series from series. Reducers consume series without producing any. As a mnemonic device, the name of each predefined OSS function begins with a letter code that
indicates the type of operation. These letters are intended to be pronounced as separate syllables.

Predefined enumerators include Elist which enumerates successive elements of a list, Evector which enumerates the elements of a vector, and Eup which enumerates the integers in a range. (The notation [...] is used to represent an OSS series.)

```
(Elist '(a b c)) => [a b c]
(Evector '#(a b c)) => [a b c]
(Eup 1 :to 3) => [1 2 3]
```

Predefined transducers include Tpositions which returns the positions of the non-null elements in a series and Tselect which selects the elements of its second argument which correspond to non-null elements of its first argument.

```
(Tpositions [a nil b c nil nil]) => [0 2 3]
(Tselect [nil T T nil] [1 2 3 4]) => [2 3]
```

Predefined reducers include Rlist which combines the elements of a series into a list, Rsum which adds up the elements of a series, Rlength which computes the
length of a series, and Rfirst which returns the first element of a series.

```
(Rlist [a b c]) => (a b c)
(Rsum [1 2 3]) => 6
(Rlength [a b c]) => 3
(Rfirst [a b c]) => a
```

As simple illustrations of how OSS functions are used, consider the following.

```
(Rsum (Evector '#(1 2 3))) => 6
(Rlist (Tpositions (Elist '(a nil b c nil)))) => (0 2 3)
```

Higher-Order OSS functions.
The OSS macro package provides a number of higher-order functions which support general classes of OSS operations. (Each of these functions end in the suffix "F", which is pronounced separately.)

For example, enumeration is supported by (EnumerateF init step test). This enumerates an OSS series of elements starting with init by repeatedly applying step. The OSS series consists of the values up to, but not including, the first value for which test is true.

Reduction is supported by (ReduceF init function items) which is analogous to the sequence function reduce. The elements of the OSS series items are combined together using function. The quantity init is used as an initial seed
value for the accumulation.

Mapping is supported by (TmapF function items) which is analogous to the sequence function map. A series is computed by applying function to each element of items.

```
(EnumerateF 3 #'1- #'minusp) => [3 2 1 0]
(ReduceF 0 #'+ [1 2 3]) => 6
(TmapF #'sqrt [4 9 16]) => [2 3 4]
```

Implicit mapping.
The OSS macro package contains a special mechanism that makes mapping particularly easy. Whenever an ordinary Lisp function is applied to an OSS series, it is automatically mapped over the elements of the OSS series. For example, in the expression below, the function sqrt is mapped over the OSS series of numbers created by Evector.

```
(Rsum (sqrt (Evector '#(4 16))))
  == (Rsum (TmapF #'sqrt (Evector '#(4 16)))) => 6
```

To a considerable extent, implicit mapping is a peripheral part of the OSS macro package---one can always use TmapF instead. However, due to the ubiquitous nature of mapping, implicit mapping is extremely convenient. As illustrated below, its key virtue is that it reduces the number of literal lambda expressions that have to be written.

```
(Rsum (expt (abs (Evector '#(2 -2 3))) 3))
  == (Rsum (TmapF #'(lambda (x) (expt (abs x) 3))
                 (Evector '#(2 -2 3)))) => 43
```

Creating OSS variables.
The OSS macro package provides two forms (letS and letS*) which are analogous to let and let*, except that they make it possible to create variables that can
hold OSS series. (The suffix "S", pronounced separately, is used to indicate primitive OSS forms.) As shown in the example below, letS can be used to bind both ordinary variables (e.g., n) and OSS variables (e.g., items).

```
(defun average (v)
  (letS* ((items (Evector v))
          (sum (Rsum items))
          (n (Rlength items)))
    (/ sum n)))

(average '#(1 2 3)) => 2
```

User-defined OSS functions.
New OSS functions can be defined by using the form defunS which is analogous to
defun. Explicit declarations are required inside defunS to indicate which arguments receive OSS series. The following example shows the definition of an
OSS function which computes the product of the numbers in an OSS series.

```
(defunS Rproduct (numbers)
    (declare (type oss numbers))
  (ReduceF 1 #'* numbers))

(Rproduct [2 4 6]) => 48
```

Restrictions on OSS expressions.
As illustrated by the examples above, OSS expressions are constructed in the same way as any other Lisp expression---i.e., OSS functions are composed together in any way desired. However, in order to guarantee that OSS expressions can always be converted into highly efficient loops, a few restrictions have to be followed. These restrictions are summarized in the beginning of Section 2 and discussed in detail in [6].

Here, it is sufficient to note that the OSS macro package is designed so that it is impossible to violate most of the restrictions. The remaining restrictions are checked by the macro package and any violations are automatically fixed. However, warning messages are issued whenever a violation
is detected, because, as discussed in the beginning of Section 2, it is often possible for the user to fix a violation in a way which is much more efficient than the automatic fix supplied by the macro package.

The best approach for programmers to take is to simply write OSS expressions without worrying about the restrictions. In this regard, it should be noted that simple OSS expressions are very unlikely to violate any of the restrictions. In particular, it is impossible for an OSS expression to violate
any of the restrictions unless it contains a variable bound by letS or defunS. When violations do occur, they can either be ignored (since they cannot lead

to
incorrect results) or dealt with on an individual basis (which is advisable since violations can lead to significant inefficiencies).

Benefits.
The benefit of OSS expressions is that they retain most of the advantages of functional programming using series, while eliminating the costs. However, given the restrictions alluded to above, the question naturally arises as to whether OSS expressions are applicable in a wide enough range of situations to be of real pragmatic benefit.

An informal study [5] was undertaken of the kinds of loops programmers actually
write. This study suggests that approximately 80% of the loops programmers write are constructed by combining a few common kinds of looping algorithms. The OSS macro package is designed so that all of these algorithms can be represented as OSS functions. As a result, it appears that approximately 80% of loops can be trivially rewritten as OSS expressions. Many more can be converted to this form with only minor modification.

Moreover, the benefits of using OSS expressions go beyond replacing individual loops. A major shift toward using OSS expressions would be a significant change in the way programming is done. At the current time, most programs contain one or more loops and most of the interesting computation in these programs occurs in these loops. This is quite unfortunate, since loops are generally acknowledged to be one of the hardest things to understand in any program. If OSS expressions were used whenever possible, most programs would not contain any loops. This would be a major step forward in conciseness, readability, verifiability, and maintainability.

Example

The following example shows what it is like to use OSS expressions in a realistic programming context. The example consists of two parts: a pair of functions which convert between sets represented as lists and sets represented as bits packed into an integer and a graph algorithm which uses the integer representation of sets.

Bit sets.
Small sets can be represented very efficiently as binary integers where each 1 bit in the integer represents an element in the set. Below, sets represented in this fashion are referred to as bit sets.

Common Lisp provides a number of bitwise operations on integers which can be used to manipulate bit sets. In particular, logior computes the union of two bit sets while logand computes their intersection.

The functions in Figure 1.1 convert between sets represented as lists and bit sets. In order to perform this conversion, a mapping has to be established between bit positions and potential set elements. This mapping is specified by
a universe. A universe is a list of elements. If a bit set b is associated with a universe u, then the ith element in u is in the set represented by b iff
the ith bit in b is 1.

For example, given the universe (a b c d e), the integer #b01011 represents the
set {a,b,d}. (By Common Lisp convention, the 0th bit in an integer is the rightmost bit.)

Given a bit set and its associated universe, the function bset->list converts the bit set into a set represented as a list of its elements. It does this by enumerating the elements in the universe along with their positions and constructing a list of the elements which correspond to 1s in the integer representing the bit set. (When no :to argument is supplied, Eup counts up forever.)

The function list->bset converts a set represented as a list of its elements into a bit set. Its second argument is the universe which is to be associated with the bit set created. For each element of the list, the function bit-position is called in order to determine which bit position should be set to 1. The function ash is used to create an integer with the correct bit set to 1. The function ReduceF is used to combine the integers corresponding to the individual elements together into a bit set corresponding to the list.

The function bit-position takes an item and a universe and returns the bit position corresponding to the item. The function operates in one of two ways depending on whether or not the item is in the universe. The first line of the function contains an OSS expression which determines the position of the item in the universe. If the item is not in the universe, the expression returns nil. (The function Rfirst returns nil if it is passed a series of length zero.)

If the item is not in the universe, the second line of the function adds the item onto the end of the universe and returns its position. The extension of the universe is done be side-effect so that it will be permanently recorded in the universe.

```
(defun bset->list (bset universe)
  (Rlist (Tselect (logbitp (Eup 0) bset) (Elist universe))))

(defun list->bset (list universe)
  (ReduceF 0 #'logior (ash 1 (bit-position (Elist list) universe))))

(defun bit-position (item universe)
  (or (Rfirst (Tpositions (eq item (Elist universe))))
    (1- (length (nconc universe (list item))))))
```

(Figure 1.1: Converting between lists and bit sets.)

Figure 1.2 shows the definition of two OSS reducers which operate on OSS series of bit sets. The first function computes the union of a series of bit sets, while the second computes their intersection.

```
(defunS Rlogior (bsets)
    (declare (type oss bsets))
  (ReduceF 0 #'logior bsets))

(defunS Rlogand (bsets)
    (declare (type oss bsets))
  (ReduceF -1 #'logand bsets))
```

(Figure 1.2: Operations on OSS series of bit sets.)

Live variable analysis.
As an illustration of the way bit sets might be used, consider the following. Suppose that in a compiler, program code is being represented as blocks of straight-line code connected by possibly cyclic control flow. The top part of Figure 1.3 shows the data structure which represents a block of code. Each block has several pieces of information associated with it. Two of these pieces of information are the blocks that can branch to the block in question and the blocks it can branch to. A program is represented as a list of blocks that point to each other through these fields.

In addition to control flow information, each structure contains information about the way variables are accessed. In particular, it records the variables that are written by the block and the variables that are used by the block (i.e., either read without being written or read before they are written). An additional field (computed by the function determine-live discussed below) records the variables which are live at the end of the block. (A variable is live if it has to be saved, because it can potentially be used by a following block.) Finally, there is a temporary data field which is used by functions (such as determine-live) which perform computations involved with the blocks.

The remainder of Figure 1.3 shows the function determine-live which, given a program represented as a list of blocks, determines the variables which are live in each block. To perform this computation efficiently, the function uses bit sets. The function operates in three steps. The first step (convert-to-bsets) looks at each block and sets up an auxiliary data structure containing bit set representations for the written variables, the used variables, and an initial guess that there are no live variables. This auxiliary structure is defined by the third form in Figure 1.3 and is stored in the temp field of the block. The integer 0 represents an empty bit set.

The second step (perform-relaxation) determines which variables are live. This is done by relaxation. The initial guess that there are no live variables in any block is successively improved until the correct answer is obtained.

The third step (convert-from-bsets) operates in the reverse of the first step. Each block is inspected and the bit set representation of the live variables is converted into a list which is stored in the live field of the block.

```
(defstruct (block (:conc-name nil))
  predecessors ;Blocks that can branch to this one.
  successors   ;Blocks this one can branch to.
  written      ;Variables written in the block.
  used         ;Variables read before written in the block.
  live         ;Variables that must be available at exit.
  temp)        ;Temporary storage location.

(defun determine-live (program-graph)
  (let ((universe (list nil)))
    (convert-to-bsets program-graph universe)
    (perform-relaxation program-graph)
    (convert-from-bsets program-graph universe))
  program-graph)

(defstruct (temp-bsets (:conc-name bset-))
  used written live)

(defun convert-to-bsets (program-graph universe)
  (letS ((block (Elist program-graph)))
    (setf (temp block)
        (make-temp-bsets
          :used (list->bset (used block) universe)
          :written (list->bset (written block) universe)
          :live 0))))

(defun perform-relaxation (program-graph)
  (let ((to-do program-graph))
    (loop
    (when (null to-do) (return (values)))
    (let* ((block (pop to-do))
           (estimate (live-estimate block)))
      (when (not (= estimate (bset-live (temp block))))
```

```
          (setf (bset-live (temp block)) estimate)
          (letS ((prev (Elist (predecessors block))))
            (pushnew prev to-do)))))))

  (defun live-estimate (block)
    (letS ((next (temp (Elist (successors block)))))
      (Rlogior (logior (bset-used next)
                       (logandc2 (bset-live next)
                             (bset-written next))))))

  (defun convert-from-bsets (program-graph universe)
    (letS ((block (Elist program-graph)))
      (setf (live block)
          (bset->list (bset-live (temp block)) universe))
      (setf (temp block) nil)))
```

(Figure 1.3: Live variable analysis.)

On each cycle of the loop in perform-relaxation, a block is examined to determine whether its live set has to be changed. To do this (see the function live-estimate), the successors of the block are inspected. Each successor needs to have available to it the variables it uses, plus the variables that are supposed to be live after it, minus the variables it writes. (The function logandc2 takes the difference of two bit sets.) A new estimate of the total set of variables needed by the successors as a group is computed by using Rlogior.

If this new estimate is different from the current estimate of what variables are live, then the estimate is changed. In addition, if the estimate is changed, perform-relaxation has to make sure that all of the predecessors of the current block will be examined to see if the new estimate for the current block requires their live estimates to be changed. This is done by adding each predecessor onto the list to-do unless it is already there. As soon as the estimates of liveness stop changing, the computation can stop.

Summary.
The function determine-live is a particularly good example of the way OSS expressions are intended to be used in two ways. First, OSS expressions are used in a number of places to express computations which would otherwise be expressed less clearly as loops or less efficiently as sequence function expressions. Second, the main relaxation algorithm is expressed as a loop. This is done, because neither OSS expressions (nor Common Lisp sequence function expressions) lend themselves to expressing the relaxation algorithm. This highlights the fact that OSS expressions are not intended to render loops entirely obsolete, but rather to provide a greatly improved method for expressing the vast majority of loops.

2. Reference Manual

This section is organized around descriptions of the various functions and forms supported by the OSS macro package. Each description begins with a header which shows the arguments and results of the function or form being described. For ease of reference, the headers are duplicated in Section 5. In
Section 5, the headers are in alphabetical order and show the page where the function or form is described.

In a reference manual like this one, it is advantageous to describe each construct separately and completely. However, this inevitably leads to presentation problems, because everything is related to everything else. Therefore, one cannot avoid referring to things which have not been discussed. The reader is encouraged to skip around in the document and to realize that more than one reading will probably be necessary in order to gain a complete understanding of the OSS macro package.

Although the following list of OSS functions is large, it should not be taken as complete. Every effort has been made to provide a wide range of useful predefined functions. However, except for a few primitive forms, all of these functions could have been defined by the user. It is hoped that users will write many more such functions. A key reason for presenting a wide array of predefined functions is to inspire users with thoughts of the wide variety of functions they can write for themselves.

Restrictions and Definitions of Terms.

As alluded to in Section 1, there are a number of restrictions which OSS expressions have to obey. The OSS macro package is designed so that all but three of these restrictions are impossible to violate with the facilities provided. As a result, the programmer need not think about these restrictions at all.

The OSS macro package checks to see that the remaining three restrictions are obeyed on an expression by expression basis and automatically fixes any violations which are detected. However, the automatic fixes are often not very efficient. As a result, it is advisable for the user to fix such violations explicitly.

Given that simple OSS expressions are very unlikely to violate any of the restrictions, and any violations which do occur are automatically fixed, it is reasonable for the reader to skip this section when first reading this manual. However, it is useful to read this section before trying to write complex OSS expressions.

The discussion below starts by defining two key terms (on-line functions and early termination) which are used to categorize the OSS functions described in the rest of this manual. The discussion then continues by briefly describing the three restrictions which can be violated. (See [6] for a complete discussion of all the restrictions.)

On-line and off-line.
Suppose that f is an OSS function which reads one or more series inputs and writes one or more series outputs. The function f is on-line [1] if it operates in the following fashion. First, f reads in the first element of each
input series, then it writes out the first element of each output series, then it reads in the second element of each input series, then it writes out the second element of each output series, and so on. In addition, f must immediately terminate as soon as any input runs out of elements. If an f is

not on-line, then it is off-line.

In the context of OSS expressions, the term on-line is generalized so that it applies to individual OSS input and output ports in addition to whole functions. An OSS port is on-line iff the processing at that port always follows the rigidly synchronized pattern described above. Otherwise, it is off-line. From this point of view, a function is on-line iff all of its OSS ports are on-line.

The prototypical example of an on-line OSS function is TmapF (which maps a function over a series). Each time it reads an input element it applies the mapped function to it and writes an output element. In contrast, the function Tremove-duplicates (which removes the duplicate elements from a series) is not on-line. Since some of the input elements do not become output elements, it is not possible for Tremove-duplicates to write an output element every time it reads an input element.

For every OSS function, the documentation below specifies which ports are on-line and which are off-line. In this regard, it is interesting to note that every function which has only one OSS port (e.g., enumerators with only one output and reducers with only one input) are trivially on-line. The only OSS functions which have off-line ports are transducers.

Early termination.
An important feature of OSS functions is the situations under which they terminate. The definition of on-line above requires that on-line functions must terminate as soon as any series input runs out of elements. If an OSS function can terminate before any of its inputs are exhausted, then it is an early terminator. The degenerate case of functions which do not have any series inputs (i.e., enumerators) is categorized by saying that enumerators are early terminators iff they can terminate.

As an example of an early terminator, consider the function TuntilF (which reads a series and returns all of the elements of that series up to, but not including, the first element which satisfies a given predicate). This function is an early terminator, because it can terminate before the input runs out of elements.

The documentation below specifies which functions are early terminators. Besides enumerators, their are only 7 OSS functions which are early terminators.

Isolation. A data flow arc delta in an OSS expression X is isolated iff it is possible to partition the functions in X into two parts Y and Y' in such a way that: delta goes from Y to Y', there is no OSS data flow from Y to Y', and there is no data flow from Y' to Y. For example, consider the OSS expression (letS ((x (f y))) (i (h x (g x)))) which corresponds to the graph in Figure 2.1. [Not shown in this textual version of the memo. See the printed version.]

The data flow arc delta4 is isolated. To show this, one merely has to partition the expression so that f, g, and h are on one side and i is on the other. The question of whether or not the other data flow arcs are isolated is more complicated to answer. If delta3 crosses a partition, then delta1 must cross this partition as well. As a result, delta3 is isolated iff delta1 carries a non-OSS value. (This is true no matter what kind of value passes over delta3 itself.) In a related situation, delta2 is isolated iff (it and therefore delta1) carries a non-OSS value. Finally, consider the arc delta1. Here there are two potential partitions to consider: one which cuts delta2 and one which cuts delta3. The data flow arc delta1 is isolated iff either it (and therefore delta2) or delta3 carries a non-OSS value.

The concept of isolation is extended to inputs and outputs as follows. An output p in an expression X is isolated iff X can be partitioned into two parts Y and Y' such that: every data flow originating on p goes from Y to Y', every other data flow from Y to Y' is a non-OSS data flow, and there is no data flow from Y' to Yf. An input q in an expression X is isolated iff X can be partitioned into two parts Y and Y' such that: the data flow terminating on q goes from Y to Y', every other data flow from Y to Y' is a non-OSS data flow, and there is no data flow from Y' to Y.

For example, in Figure 2.1, the outputs of f and h are isolated as is the input
of i. The input and output of g are isolated iff f computes a non-OSS value. The inputs of h are isolated iff the data flow arcs terminating on them are isolated.

Non-OSS data flows must be isolated.
In order for an OSS expression to be reliably converted into a highly efficient loop, every non-OSS data flow in it must be isolated. As an example of an expression where this is not true, consider the following. In this expression, the data flow implemented by the variable total is not isolated.

```
  (letS* ((nums (Evector '#(3 2 8)))            ;Signals warning 16
          (total (ReduceF 0 #'+ nums)))
    (Rvector (/ nums total))) => #(3/13 2/13 8/13)
```

(The basic problem here is that while the elements created by Evector are being
used to compute total, they all have to be saved so that they can be used again
later in order to perform the indicated divisions. Eliminating the need for such storage is the key source of efficiency underlying OSS expressions.)

Off-line OSS ports must be isolated.
In order for an OSS expression to be reliably converted into a highly efficient loop, every off-line port must be isolated. As an example of an expression which has an off-line output which is not isolated, consider the following. In this expression, the data flow implemented by the variable positions is not isolated.

```
  (letS* ((keys (Elist list))                   ;Signals warning 17.1
          (positions (Tpositions keys)))
    (Rlist (list positions keys)))
```

(The basic problem here is that since Tpositions skips null elements of the input, Tpositions sometimes has to read several input elements before it can produce the next output element. This forces an unpredictable number of elements of keys to be saved so that they can be used later when creating lists. As above, eliminating the need for such storage is the main goal of OSS expressions.)

Code copying.
If an OSS expression violates either of the above restrictions, the OSS macro packaged automatically fixes the problem by copying code until the data flow or port in question becomes isolated. For instance, the example above of an OSS expression in which a non-OSS data flow is not isolated is fixed as follows.

```
(letS* ((nums (Evector '#(3 2 8)))
        (total (ReduceF 0 #'+ (Evector '#(3 2 8)))))
  (Rvector (/ nums total))) => #(3/13 2/13 8/13)
```

Even though the problem has been automatically fixed, the OSS macro package issues a warning message. This is done for two reasons. First, if side-effects (e.g., input or output) are involved, the code copying that was performed may not be correctness preserving. Second, large amounts of code sometimes have to be copied and that can introduce large amounts of extra computation.

A major goal of OSS expressions is ensuring that expressions which look simple to compute actually are simple to compute. Automatically introducing large amounts of additional computation without the programmer's knowledge would violate this goal. At the very least, issuing warning messages makes programmers aware of what is expensive to compute and what is not. Looked at from a more positive perspective, it encourages them to think of ways to compute what they want without code copying being required.

For instance, consider the example above of an OSS expression in which an off-line port is not isolated. It might be the case that the programmer knows that list does not contain any null elements and that Tpositions is therefore merely being used to enumerate what the positions of the elements are. In this situation, the expression can be fixed as follows, which does not require any code copying. (The key insight here is that the positions do not actually depend on the values in the list.)

```
(let ((list '(a b c)))
  (letS* ((keys (Elist list))
          (positions (Eup 0)))
    (Rlist (list positions keys)))) => ((0 a) (1 b) (2 c))
```

(It is interesting to note that if an expression is a tree (as opposed to a graph as in Figure 2.1), then every data flow arc and every port is isolated. This is why OSS expressions which do not contain variables bound by letS, lambdaS, or defunS cannot violated either of the isolation restrictions. This is also why code copying can always fix any violation---code copying can convert any graph into a tree.)

On-line subexpressions.
The two isolation restrictions above permit a divide and conquer approach to the processing of OSS expressions. If an OSS expression obeys the isolation restrictions, then it can be repeatedly partitioned until all of the data flow in each subexpression goes from an on-line output to an on-line input. The subexpressions which remain after partitioning are referred to as on-line subexpressions.

Termination points.
The functions in each on-line subexpression can be divided into two classes: those which are termination points and those which are not. A function is a termination point if it can terminate before any other function in the subexpression terminates. There are two reasons for functions being termination points. Functions which are early terminators are always termination points. In addition, any function which reads an OSS series which comes from a different on-line subexpression is a termination point.

Data flow paths between termination points and outputs.
In order for an OSS expression to be reliably converted into a highly

efficient loop, it must be the case that within each on-line subexpression, there is a data flow path from each termination point to each output. As an example of an OSS expression for which this property does not hold, consider the following. Partitioning divides this expression into two on-line subexpressions, one containing list and one containing everything else. In the large on-line subexpression, the two instances of Evector are termination points. The program violates the property above, because there is no data flow path from the termination point (Evector weight-vector) to the output of (Rvector squares).

```
(defun weighted-squares (value-vector weight-vector)
  (letS* ((values (Evector value-vector))        ;Signals warning 18
          (weights (Evector weight-vector))
          (squares (* values values))
          (weighted-squares (* squares weights)))
    (list (Rvector squares) (Rvector weighted-squares))))

(weighted-squares #(1 2 3) #(2 3 4)) => (#(1 4 9) #(2 12 36))
(weighted-squares #(1 2) #(2 3 4)) => (#(1 4) #(2 12))
(weighted-squares #(1 2 3) #(2 3)) => (#(1 4 9) #(2 12))
```

(The basic problem here is that if the number of elements in value-vector is greater than the number of elements in weight-vector, the computation of squares has to continue even after the computation of weighted-squares has been completed. This kind of partial continuing evaluation in a single on-line subexpression is not supported by the OSS macro package, because it was judged that it requires too much overhead in order to control what gets evaluated when.)

When an OSS expression violates the restriction above, the violation is automatically fixed by applying code copying. It is impossible for an on-line subexpression to violate the restriction unless it computes two different outputs. Code copying can always be used to break the subexpression in question into two parts each of which computes one of the outputs. Unfortunately, this can require a great deal of code to be copied. There are two basic approaches which can be used to fix a violation much more efficiently: reducing the number of termination points and increasing the connectivity between termination points and outputs.

The easiest way to decrease the number of termination points is to replace early terminators by equivalent operations which are not early terminators. If an early terminator is not an enumerator, then this can always be done without difficultly. (The documentation below describes a non-early variant for each early terminating transducer and reducer.) If multiple enumerators are the problem (as in the example above) decreasing the number of termination points is usually not practical. However, sometimes an enumerator which terminates can be replaced by an enumerator which never terminates.

The connectivity between termination points and outputs can be increased by using the function Tcotruncate. This is the preferred way to fix the problem in the example above.

General Information

Before discussing the individual OSS functions in detail, a few general comments are in order. First, all of the OSS functions and forms are defined in the package OSS. To make these names easily accessible, use the package OSS (i.e., evaluate (use-package "OSS")). If this is not done, the names will have to be prefixed with "oss:" when they are used.

Naming conventions.
The names of the various OSS functions and forms follow a strict naming convention. The first letter of an OSS function name indicates the type of function as shown below. The letter codes are written in upper case in this document (case does not matter to Common Lisp) and each letter is intended to be pronounced as a separate syllable.

E Enumerator.
T Transducer.
R Reducer.

The last letter of each OSS special form is "S". In general, this indicates that the form is primitive in the sense that it could not be defined by the user. Some OSS functions end in the letter "F". This is used to indicate that the function is a higher-order function which takes functions as arguments.

The naming convention has two advantages: one trivial but vital and the other more fundamentally useful. First, many of the OSS functions are very similar to standard Common Lisp sequence functions. As a result, it makes sense to give them similar names. However, it is not possible to give them exactly the same names without redefining the standard functions. The naming convention allows the names to be closely related in a predictable way without making the names unreasonably long.

Second, the naming convention highlights several properties of OSS functions which make it easier to read and understand OSS expressions. In particular, the prefixes highlight the places where series are created and consumed.

The names of arguments and results of OSS functions are also chosen following naming conventions. First, all of the names are chosen in an attempt to indicate type restrictions (e.g., number indicates that an argument must be a number; item indicates that there is no type restriction). Plural names are used iff the value in question is an OSS series (e.g., numbers indicates an OSS
series of numbers; items indicates an OSS series of unrestricted values). The name of a series input or output begins with "O" iff it is off-line.

OSS series. Two general points about OSS series are worthy of note. First, like Common Lisp sequences, OSS series use zero-based indexing (i.e., the first element is the 0th element). Second, unlike Common Lisp sequences, OSS series can be unbounded in length.

Tutorial mode.
A prominent feature of the various descriptions is that they contain many examples. These examples contain large numbers of OSS series as inputs and outputs. In the interest of brevity, the notation [...] is used to indicate a literal OSS series. If the last entry in a literal OSS series is an ellipsis, this indicates that the OSS series is unbounded in length.

```
[1 2 3]
[a b (c d)]
[T nil T nil ...]
```

The notation [...] is not supported by the OSS macro package. It would be straightforward to do so by using set-macro-character. Perhaps even better, one could use set-dispatch-macro-character to support a notation #[...]

analogous to #(...). However, although literal series are very useful in the examples below, experience suggests that literal series are seldom useful when writing actual programs. Inasmuch as this is the case, it was decided that it was unwise to use up one of the small set of characters which are available for user-defined reader macros or user-defined # dispatch characters.

Many of the examples show OSS expressions returning OSS series as their values. However, one should not take this literally. If these examples are typed to Common Lisp as isolated expressions, they will not return any values. This is so, because the OSS macro package does not allow complete OSS expressions to return OSS series. The examples are intended to show what would be returned if the example expressions were nested in larger expressions.

```
oss-tutorial-mode  &optional (T-or-nil T) => state-of-tutorial-mode
```

The above not withstanding, the OSS macro package provides a special tutorial mode in which the notation [...] is supported and OSS expressions can return (potentially unbounded) OSS values. However, these values still cannot be stored in ordinary variables. This mode is entered by calling the function oss-tutorial-mode with an argument of T. Calling the function with an argument of nil turns tutorial mode off.

Using tutorial mode, it is possible to directly duplicate the examples shown below. However, tutorial mode is very inefficient. What is worse, tutorial mode introduces non-correctness-preserving changes in OSS expressions. (For example, in order to correctly duplicate the examples that illustrate error messages about non-terminating expressions and the fact that OSS series are not actually returned by complete OSS expressions, tutorial mode must be turned off.) All in all, it is important that tutorial mode not be used as anything other than an educational aid.

OSS functions are actually macros.
Every OSS function is actually a macro. As a result, OSS functions cannot be funcall'ed, or apply'ed. When the user defines new OSS functions, they must be defined before the first time they are used. Also, when an OSS function takes keyword arguments, the keywords must be literals. They cannot be expressions which evaluate to keywords at run time.

Finally, the macro expansion processing associated with OSS expressions is relatively time consuming. In order to avoid this overhead during the running of a user program, it is important that programs containing OSS expressions be compiled rather than run interpretively.

A minor advantage of the fact that everything in the OSS macro package is a macro is that once a program which uses the macro package is compiled, the compiled program can subsequently be run without having to load the OSS macro package.

A more important advantage of the fact that everything in the OSS macro package is a macro is that quoted macro names can be used as functional arguments to higher-order OSS functions. (In contrast, quoted macro names cannot be used as functional arguments to higher-order Common Lisp functions such as reduce.) Although this may appear to be a minor benefit, it is actually quite useful.

Enumerators

Enumerators create OSS outputs based on non-OSS inputs. There are two basic kinds of enumerators: ones that create an OSS series based on some formula (e.g., enumerating a sequence of integers) and ones that create an OSS series containing the elements of an aggregate data structure (e.g., enumerating the elements of a list). All the predefined enumerators are on-line. In general, they are all early terminators. However, as noted below, in some situations, some enumerators produce unbounded outputs and are not early terminators.

Eoss &rest expr-list => items

The expr-list consists of zero or more expressions. The function Eoss creates an OSS series containing the values of these expressions. Every expression in expr-list is evaluated before the first output element is returned.

```
(Eoss 1 'a 'b) => [1 a b]
(Eoss) => []
```

To get the effect of delaying the evaluation of individual elements until they are needed, it is necessary to define a special purpose enumerator which computes the individual items as needed. However, due to the control overhead required, this is seldom worthwhile.

It is possible for the expr-list to contain an instance of :R. (This must be a literal instance of :R, not an expression which evaluates to :R.) If this is the case, then Eoss produces an unbounded OSS series analogous to a repeating decimal number. The output consists of the values of the expressions preceding the :R followed by an unbounded number of repetitions of the values following the :R, if there are any such values. (In this situation, Eoss is not an early terminator.)

```
(Eoss 1 'a :R 'b 'c) => [1 a b c b c b c ...]
(Eoss T :R nil) => [T nil nil nil ...]
(Eoss 1 :R) => [1]
(Eoss :R 1) => [1 1 1 ...]
```

Eup &optional (start 0) &key (:by 1) :to :below :length => numbers

This function is analogous to the Loop macro [2] numeric iteration clause. It creates an OSS series of numbers starting with start and counting up by :by. The argument start is optional and defaults to integer 0. The keyword argument :by must always be a positive number and defaults to integer 1.

There are four kinds of end tests. If :to is specified, stepping stops at this number. The number :to will be included in the OSS series iff (- :to start) is a multiple of :by. If :below is specified, things operate exactly as if :to were specified except that the number :below is never included in the OSS series. If :length is specified, the OSS series has length :length. It must be the case that :length is a non-negative integer. If :length is positive, the last element of the OSS series will be (+ start (* :by (1- :length))). If none of the termination arguments are specified, the output has unbounded length. (In this situation, Eup is not an early terminator.) If more than one termination argument is specified, it is an error.

```
(Eup :to 4) => [0 1 2 3 4]
(Eup :to 4 :by 3) => [0 3]
(Eup 1 :below 4) => [1 2 3]
(Eup 4 :length 3) => [4 5 6]
(Eup) => [0 1 2 3 4 ...]
```

As shown in the following example, Eup does not assume that the numbers being enumerated are integers.

```
(Eup 1.5 :by .1 :length 3) => [1.5 1.6 1.7]
```

Edown &optional (start 0) &key (:by 1) :to :above :length => numbers

The function Edown is analogous to Eup, except that it counts down instead of up and uses the keyword :above instead of :below.

```
(Edown :to -4) => [0 -1 -2 -3 -4]
(Edown :to -4 :by 3) => [0 -3]
(Edown 1 :above -4) => [1 0 -1 -2 -3]
(Edown 4 :length 3) => [4 3 2]
(Edown) => [0 -1 -2 -3 -4 ...]
(Edown -1.5 :by .1 :length 3) => [-1.5 -1.6 -1.7]
```

Esublists list &optional (end-test #'endp) => sublists

This function creates an OSS series containing the successive sublists of list. The end-test must be a function from objects to boolean values (i.e., to null/non-null). It is used to determine when to stop the enumeration. Successive cdrs are returned up to, but not including, the first one for which end-test returns non-null.

```
(Esublists '(a b c)) => [(a b c) (b c) (c)]
(Esublists '(a b . c) #'atom) => [(a b . c) (b . c)]
```

The default end-test (#'endp) will cause Esublists to blow up if list contains a non-list cdr. More robust enumeration can be obtained by using the end-test #'atom as in the second example above. The assumption that list will end with nil is used as the default case, because the assumption sometimes allows programming errors to be detected closer to their sources.

Elist list &optional (end-test #'endp) => elements

This function creates an OSS series containing the successive elements of list. It is closely analogous to Esublists as shown below. In particular, end-test has the same meaning and the same caveats apply.

```
(Elist '(a b c)) => [a b c]
(Elist '()) => []
(Elist '(a b . c) #'atom) => [a b]
(Elist list) == (car (Esublists list))
```

The value returned by Elist can be used as a destination for alterS.

```
(let ((list '(a b c)))
  (alterS (Elist (cdr list)) (Eup))
  list) => (a 0 1)
```

Ealist alist &optional (test #'eql) => keys values

This function returns two OSS series containing keys and their associated values. The first element of keys is the key in the first entry in alist, the first element of values is the value in the first entry, and so on. The alist must be a proper list ending in nil and each entry in alist must be a cons cell or nil. Like assoc, Ealist skips entries which are nil and entries which have the same key as an earlier entry. The test argument is used to determine when two keys are the same.

```
(Ealist '((a . 1) () (a . 3) (b . 2))) => [a b] [1 2]
(Ealist nil) => [] []
```

Both of the series returned by Ealist can be used as destinations for alterS. (In analogy with multiple-value-bind, letS can be used to bind both values returned by Ealist.)

```
(let ((alist '((a . 1) (b . 2))))
  (letS (((key val) (Ealist alist)))
    (alterS key (list key))
    (alterS val (1+ val)))
  alist) => '(((a) . 2) ((b) . 3))
```

The OSS function Ealist is forced to perform a significant amount of computation in order to check that no duplicate keys or null entries are being enumerated. In a situation where it is known that no duplicate keys or null entries exist, it is much more efficient to use Elist as shown below.

```
(letS* ((e (Elist '((a . 1) (b . 2))))
        (keys (car e))
        (values (cdr e)))
  (Rlist (list keys values))) => ((a 1) (b 2))
```

Eplist plist => indicators values

This function returns two OSS series containing indicators and their associated values. The first element of indicators is the first indicator in the plist, the first element of values is the associated value, and so on. The plist argument must be a proper list of even length ending in nil. In analogy with the way get works, if an indicator appears more than once in plist, it (and its value) will only be enumerated the first time it appears. (Both of the OSS series returned by Eplist can be used as destinations for alterS.)

```
(Eplist '(a 1 a 3 b 2)) => [a b] [1 2]
(Eplist nil) => [] []
```

The OSS function Eplist has to perform a significant amount of computation in order to check that no duplicate indicators are being enumerated. In a situation where it is known that no duplicate indicators exist, it is much more efficient to use EnumerateF as shown below.

```
(letS* ((e (EnumerateF '(a 1 b 2) #'cddr #'null))
        (indicators (car e))
        (values (cadr e)))
  (Rlist (list indicators values))) => ((a b) (1 2))
```

Etree tree &optional (leaf-test #'atom) => nodes

This function creates an OSS series containing all of the nodes in tree. The function assumes that tree is a tree built of lists, where each node is a list and the elements in the list are the children of the node. The function Etree does not assume that the node lists end in nil; however, it ignores any non-list cdrs. (This behavior increases the utility of Etree when it is used to scan Lisp code.) The nodes in the tree are enumerated in preorder (i.e., first the root is output, then the nodes in the tree which is the first child of the root is enumerated in full, then the nodes in the tree which is the second child of the root is enumerated in full, etc.).

The leaf-test is used to decide which elements of the tree are leaves as opposed to internal nodes. Failure of the test should guarantee that the element is a list. By default, leaf-test is #'atom. This choice of test categorizes nil as a leaf rather than as a node with no children.

The function Etree assumes that tree is a tree as opposed to a graph. If tree is a graph instead of a tree (i.e. some node has more than one parent), then this node (and its descendants) will be enumerated more than once. If the tree is a cyclic graph, then the output series will be unbounded in length.

```
(Etree 'd) => [d]
(Etree '((c) d)) => [((c) d) (c) c d]
(Etree '((c) d)
     #'(lambda (e)
         (or (atom e) (atom (car e))))) => [((c) d) (c) d]
```

Efringe tree &optional (leaf-test #'atom) => leaves

This enumerator is the same as Etree except that it only enumerates the leaves of the tree, skipping all internal nodes. The logical relationship between Efringe and Etree is shown in the first example below. However, Efringe is implemented more efficiently than this example would indicate.

```
(Efringe tree) == (TselectF #'atom (Etree tree))
(Efringe 'd) => [d]
(Efringe '((c) d)) => [c d]
(Efringe '((c) d)
       #'(lambda (e)
           (or (atom e) (atom (car e))))) => [(c) d]
```

The value returned by Efringe can be used as a destination for alterS. However, if the entire tree is a leaf and gets altered, this will have no side-effect on the tree as a whole. In addition, altering a leaf will have no effect on the leaves enumerated. In particular, if a leaf is altered into a subtree, the leaves of this subtree will not get enumerated.

```
(let ((tree '((3) 4)))
  (letS ((leaf (Efringe tree)))
    (if (evenp leaf) (alterS leaf (- leaf))))
  tree) => ((3) -4)
```

Evector vector &optional (indices (Eup)) => elements

This function creates an OSS series of the elements of a one-dimensional array. If indices assumes its default value, Evector enumerates all of the elements of vector in order.

```
(Evector "BAR") => [#\B #\A #\R]
(Evector "") => []
```

Looked at in greater detail, Evector enumerates the elements of vector which are indicated by the elements of the OSS series indices. The indices must be non-negative integers, however, they do not have to be in order. Enumeration stops when indices runs out, or an index greater than or equal to the length of vector is encountered. One can use Eup to create an index series which picks out a section of vector. (Since Evector takes in an OSS series it is technically a transducer, however, it is on-line and is an enumerator in spirit.)

```
(Evector '#(b a r) (Eup 1 :to 2)) => [a r]
(Evector "BAR" [0 2 1 1 4 1]) => [#\B #\R #\A #\A]
```

The value returned by Evector can be used as a destination for alterS.

```
(let ((v "FOOBAR"))
  (alterS (Evector v (Eup 2 :to 4)) #\-) v) => "FO---R"
```

Esequence sequence &optional (indices (Eup)) => elements

The function Esequence is the same as Evector except that it will work on any Common Lisp sequence. However, since it has to determine the type of sequence at run-time, it is much less efficient than either Elist or Evector. (The value returned by Esequence can be used as a destination for alterS.)

```
(Esequence '(b a r)) => [b a r]
(Esequence '#(b a r)) => [b a r]
```

Ehash table => keys values

This function returns two OSS series containing keys and their associated values. The first element of keys is the key of the first entry, the first element of values is the value in the first entry, and so on. (There are no guarantees as to the order in which entries will be enumerated.)

```
(Ehash (let ((h (make-hash-table)))
         (setf (gethash 'color h) 'brown)
         (setf (gethash 'name h) 'fred)
        h)) => [color name] [brown fred] ;in some order
```

In the pure Common Lisp version of the OSS macro package, Ehash is rather inefficient, because Common Lisp does not provide incremental support for scanning the elements of a hash table. However, in the Symbolics Common Lisp version of the OSS macro package, Ehash is quite efficient.

Esymbols &optional (package *package*) => symbols

This function creates an OSS series of the symbols in package (which defaults to the current package). (There are no guarantees as to the order in which symbols will be enumerated.)

```
(Esymbols) => [foo bar baz ... zot] ;in some order
```

In the pure Common Lisp version of the OSS macro package, Esymbols is rather inefficient, because Common Lisp does not provide incremental support for scanning the symbols in a package. However, in the Symbolics Common Lisp version of the OSS macro package, Esymbols is quite efficient.

Efile name => items

This function creates an OSS series of the items written in the file named name. The function combines the functionality of with-open-file with the action of reading from the file (using read). It is guaranteed that the file will be closed correctly, even if an error occurs. As an example of using Efile, assume that the forms (a), (1 2), and T have been written into the file "test.lisp".

```
(Efile "test.lisp") => [(a) (1 2) T]
```

EnumerateF init step &optional test => items

The higher-order function EnumerateF is used to create new kinds of enumerators. The init must be a value of some type T1. The step argument must be a non-OSS function from T1 to T1. The test argument (if present) must be a non-OSS function from T1 to boolean.

Suppose that the series returned by EnumerateF is S. The first output element S_0 has the value S_0=init. For subsequent elements, S_i=step(S_i-1).

If the test is present, the output consists of elements up to, but not including, the first element for which test(S_i) is true. In addition, it is guaranteed that step will not be applied to the element for which test is true. If there is no test, then the output series will be of unbounded length. (In this situation, EnumerateF is not an early terminator.)

```
(EnumerateF '(a b c d) #'cddr #'null) => [(a b c d) (c d)]
(EnumerateF '(a b c d) #'cddr) => [(a b c d) (c d) nil nil ...]
(EnumerateF list #'cdr #'null) == (Esublists list)
```

If there is no test, then each time an element is output, the function step is applied to it. Therefore, it is important that other factors in an expression cause termination before EnumerateF computes an element which step cannot be applied to. In this regard, it is interesting that the following equivalence is almost, but not quite true. The difference is that including the test argument in the call on EnumerateF guarantees that step will not be applied to the element which fails test, while the expression using TuntilF guarantees that it will.

```
(TuntilF test (EnumerateF init step)) not= (EnumerateF init step test)
```

Enumerate-inclusiveF init step test => items

The higher-order function Enumerate-inclusiveF is the same as EnumerateF except that the first element for which test is true is included in the output. As with EnumerateF, it is guaranteed that step will not be applied to the element for which test is true.

```
(Enumerate-inclusiveF '(a b) #'cddr #'null) => [(a b) ()]
```

On-Line Transducers

Transducers compute OSS series from OSS series and form the heart of most OSS expressions. This section and the next one present the predefined transducers that are on-line (i.e., all of their inputs and outputs are on-line). These transducers are singled out because they can be used more flexibly than the transducers which are off-line. In particular, it is impossible to violate the off-line port isolation restriction without using an off-line transducer.

Tprevious items &optional (default nil) (amount 1) => shifted-items

This function creates a series which is shifted right amount elements. The input amount must be a positive integer. The shifting is done by inserting amount copies of default before items and discarding amount elements from the end of items. The output is always the same length as the input.

```
(Tprevious [a b c]) => [nil a b]
(Tprevious [a b c] 'z) => [z a b]
(Tprevious [a b c] 'z 2) => [z z a]
(Tprevious []) => []
```

The word previous is used as the root for the name of this function, because the function is typically used to access previous values of a series. An example of Tprevious used in this way is shown in conjunction with Tuntil below.

To insert some amount of stuff in front of a series without losing any of the elements off the end, use Tconcatenate as shown below.

```
(Tconcatenate [z z] [a b c]) => [z z a b c]
```

Tlatch items &key :after :before :pre :post => masked-items

This function acts like a latch electronic circuit component. Each input element causes the creation of a corresponding output element. After a specified number of non-null input elements have been encountered, the latch is triggered and the output mode is permanently changed.

The :after and :before arguments specify the latch point. The latch point is just after the :after-th non-null element in items or just before the :before-th non-null element. If neither :after nor :before is specified, an :after of 1 is assumed. If both are specified, it is an error.

If a :pre is specified, every element prior to the latch point is replaced by this value. If a :post is specified, this value is used to replace every element after the latch point. If neither is specified, a :post of nil is assumed.

```
(Tlatch [nil c nil d e]) => [nil c nil nil nil]
(Tlatch [nil c nil d e] :before 2 :post T) => [nil c nil T T]
(Tlatch [nil c nil d e] :before 2 :pre 'z) => [z z z d e]
```

As a more realistic example of using Tlatch, suppose that a programmer wants to write a program get-codes which takes in a list and returns a list of all of the numbers which appear in the list after the second number in the list.

```
(defun get-codes (list)
  (letS ((elements (Elist list)))
    (Rlist (Tselect (Tlatch (numberp elements) :after 2 :pre nil)
                    elements))))

(get-codes '(a b 3 4 c d 5 e 6 f)) => (5 6)
```

Tuntil bools items => initial-items

This function truncates an OSS series of elements based on an OSS series of boolean values. The output consists of all of the elements of items up to, but not including, the first element which corresponds to a non-null element of bools. That is to say, if the first non-null value in bools is the mth, the output will consist of all of the elements of items up to, but not including, the mth. (The effect of including the mth element in the output can be obtained by using Tprevious as shown in the last example below.) In addition, the output terminates as soon as either input runs out of elements
even if a non-null element of bools has not been encountered.

```
(Tuntil [nil nil T nil T] [1 2 -3 4 -5]) => [1 2]
(Tuntil [nil nil T nil T] [1]) => [1]
(Tuntil (Eoss :R nil) (Eup)) => [0 1 2 ...]
(Tuntil [nil nil T nil T] (Eup)) => [0 1]
(letS ((x [1 2 -3 4 -5]))
  (Tuntil (minusp x) x)) => [1 2]
(letS ((x [1 2 -3 4 -5]))
  (Tuntil (Tprevious (minusp x)) x)) => [1 2 -3]
```

If the items input of Tuntil is such that it can be used as a destination for alterS, then the output of Tuntil can be used as a destination for alterS.

```
(letS* ((list '(a b 10 c))
        (x (Elist list))
        (y (Tuntil (numberp x) x)))
  (alterS y (list y))
  list) => ((a) (b) 10 c)
```

TuntilF pred items => initial-items

This function is the same as Tuntil except that it takes a functional argument instead of an OSS series of boolean values. The non-OSS function pred is mapped over items in order to obtain a series of boolean values. (Like Tuntil, TuntilF is can be used as a destination of alterS if items can.) The basic relationship between TuntilF and Tuntil is shown in the last example below.

```
(TuntilF #'minusp [1 2 -3 4 -5]) => [1 2]
(TuntilF #'minusp [1]) => [1]
(TuntilF #'minusp (Eup)) => [0 1 2 ...]
(TuntilF pred items)
  == (letS ((var items)) (Tuntil (TmapF pred var) var))
```

The functions Tuntil and TuntilF are both early terminators. This can sometimes lead to conflicts with the restriction that within each on-line subexpression, there must be a data flow path from each termination point to each output. To get the same effect without using an early terminator use Tselect of Tlatch as shown below.

```
(Tuntil bools items)
  == (Tselect (not (Tlatch bools :post T)) items)

(TuntilF #'pred items)
  == (Tselect (not (Tlatch (pred items) :post T)) items)
```

TmapF function &rest items-list => items

The higher-order function TmapF is used to create simple kinds of on-line transducers. Its arguments are a single function and zero or more OSS series. The function argument must be a non-OSS function which is compatible with the number of input series and the types of their elements.

A single OSS series is returned. Each element of this series is the result of applying function to the corresponding elements of the input series. (That is to say, if TmapF receives a single input series R it will return a single output S such that S_i=function(R_i).) The length of the output is the same as the length of the shortest input. If there are no bounded series inputs (e.g., if there are no series inputs), then TmapF will generate an unbounded OSS series.

```
(TmapF #'+ [1 2 3] [4 5]) => [5 7]
(TmapF #'sqrt []) => []
(TmapF #'gensym) => [#:G003 #:G004 #:G005 ...]
```

TscanF {init} function items => results

The higher-order function TscanF is used to create complex kinds of on-line transducers. (The name is borrowed from APL.) The init argument (if present) must be a non-OSS value of some type T1. The function argument must be a binary non-OSS function from T1 and some type T2 to T1. The items argument must be an OSS series whose elements are of type T2. If the init argument is not present than T1 must equal T2.

The function argument is used to compute a series of accumulator values of type T1 which is returned as the output of TscanF. The output is the same length as the series input and consists of the successive accumulator values.

Suppose that the series input to TscanF is R and the output is S. The basic relationship between the output and the input is that S_i=function(S_i-1,R_i). If the init argument is specified, it is used as an initial value of the accumulator and the first output element S_0 has the value S_0=function(init,R_0). Typically, but not necessarily, init is chosen so that
it is a left identity of function. If that is the case, then S_0=R_0. It is important to remember that the elements of items are used as the second argument of function. The order of arguments is chosen to highlight this fact.

```
(TscanF 0 #'+ [1 2 3]) => [1 3 6]
(TscanF 10 #'+ [1 2 3]) => [11 13 16]
(TscanF nil #'cons [a b]) => [(nil . a) ((nil . a) . b)]
(TscanF nil #'(lambda (state x) (cons x state)) [a b]) => [(a) (b a)]
```

If the init argument is not specified, then the first element of the output is computed differently from the succeeding elements and S_0=R_0. (If function is cheap to evaluate, TscanF runs more efficiently if it is provided with an init argument.) One situation where one typically has to leave out the init argument is when function does not have a left identity element as in the last example below.

```
(TscanF #'+ [1 2 3]) => [1 3 6]
(TscanF #'max [1 3 2]) => [1 3 3]
```

An interesting example of a scanning process is the operation of proration. In this process, a total is divided up and allocated between a number of categories. The allocation is done based on percentages which are associated with the categories. (For example, some number of packages might be divided up between a number of people.) One might think that this could be done straightforwardly by multiplying the total by each of the percentages. Unfortunately, this mapping approach does not work.

The proration problem is more complex than it first appears. Typically, there is a limit to the divisibility of the total (e.g., when a group of packages is divided up, the individual packages cannot be subdivided). This means that rounding must be performed each time the total is multiplied by a percentage. In addition, it is usually important that the total be allocated exactly---i.e., that the sum of the allocations be exactly equal to the total, rather than being one more or one less. Scanning is required in order to make sure that things come out exactly right.

As a concrete example of proration, suppose that 99 packages need to be allocated among three people based on the percentages 35%, 45%, and 20%. Assuming that the percentages and the number of packages are all represented as integers, simple mapping would lead to the incorrect result below in which the allocations add up to 100 instead of 99.

```
(prognS (round (/ (* 99 [35 45 20]) 100))) => [35 45 20]
```

The transducer Tprorate below solves the proration problem by using TscanF. It takes in a total and an OSS series of percentages and returns an OSS series of allocations. The basic action of the program is to multiply each percentage by the total. However, it also keeps track of how much of the total has been allocated. When the last percentage is encountered, the allocation is set to be everything which remains to be allocated. (This can cause a significant distortion in the final allocation, but it guarantees that the allocations will always add up to the total no matter what has happened with rounding along the way.) In order to determine when the last percentage is being encountered, the program keeps track of how much percentage has been accounted for and assumes that the percentages always add up to 100.

```
(defun prorate-step (state percent)
  (let* ((total (second state))
         (unallocated (third state))
         (unused-percent (fourth state))
         (allocation (if (= percent unused-percent) unallocated
                         (round (/ (* total percent) 100)))))
    (setf (first state) allocation)
    (setf (third state) (- unallocated allocation))
    (setf (fourth state) (- unused-percent percent))
    state))

(defunS Tprorate (total percents)
    (declare (type oss percents))
  (car (TscanF (list 0 total total 100) #'prorate-step percents)))

(Tprorate 99 [35 45 20]) => [35 45 19]
```

An interesting aspect of the function Tprorate is that the state manipulated by the scanned function prorate-step has four parts: an allocation, the total, the unallocated portion of the total, and the remaining percentage not yet allocated. This illustrates the fact that TscanF can be used with complex state objects. (The same is true of EnumerateF and ReduceF.) However, it also illustrates that accessing the various parts of a complex state is awkward and inefficient.

Fortunately, it is often possible to get around the need for a complex state object by using a compound OSS expression. For the example of proration, this can be done as shown below. Simple mapping is combined with two scans which keep track of cumulative values. An implicitly mapped test is used to make sure that things come out right on the last step. (The function Tprevious is used to access the previous value of the series unallocated.)

```
(defunS Tprorate-multi-state (total percents)
    (declare (type oss percents))
  (letS* ((allocation (round (/ (* percents total) 100)))
```

```
          (unallocated (TscanF total #'- allocation))
          (unused-percent (TscanF 100 #'- percents)))
      (if (zerop unused-percent)
        (Tprevious unallocated total)
        allocation)))
```

Cotruncation

A key feature of every on-line transducer is that it terminates as soon as any input runs out of elements. Put another way, the output is never longer than the shortest input. (If the transducer is also an early terminator, then the output can be shorter than the shortest input, otherwise it must be the same length as the shortest input.) This effect is referred to as cotruncation, because it acts as if each input had been truncated to the length of the shortest input. If several enumerators and on-line transducers are combined together into an OSS expression, cotruncation will typically cause all of the series produced by the enumerators to be truncated to the same length. For example, in the expression below, all of the series (including the unbounded series produced by Eup) are truncated to a length of two.

```
  (Rlist (* (+ (Eup) [4 5]) [1 2 3])) => (4 12)
```

Tcotruncate items &rest more-items => initial-items &rest more-initial-items

It is occasionally important to specify cotruncation explicitly. This can be done with the function Tcotruncate whose only action is to force all of the outputs to be of the same length. (If any of the inputs of Tcotruncate are such that they can be used as destinations of alterS, then the corresponding outputs of Tcotruncate can be used as destinations of alterS.)

```
  (Tcotruncate [1 2 -3 4 -5] [10]) => [1] [10]
  (Tcotruncate (Eup) [a b]) => [0 1] [a b]
  (Tcotruncate [a b] []) => [] []
```

An important feature of Tcotruncate is that it has a powerful interaction with the requirement that within each on-line subexpression, there must be a data flow path from each termination point to each output. Consider the function weighted-squares below. This program is intended to take a vector of values and a vector of weights and return a list of two vectors: the squares of the values and the squares multiplied by the weights. The program violates the requirement above, because there is no data flow path from (Evector weight-vector) to (Rvector squares).

```
  (defun weighted-squares (value-vector weight-vector)
    (letS* ((values (Evector value-vector))       ;Signals warning 18
          (weights (Evector weight-vector))
          (squares (* values values))
          (weighted-squares (* squares weights)))
      (list (Rvector squares) (Rvector weighted-squares))))

  (weighted-squares #(1 2 3) #(2 3 4)) => (#(1 4 9) #(2 12 36))
  (weighted-squares #(1 2) #(2 3 4)) => (#(1 4) #(2 12))
  (weighted-squares #(1 2 3) #(2 3)) => (#(1 4 9) #(2 12))
```

It might be the case that the programmer knows that value-vector and weight-vector always have the same length. (Or it might be the case that he wants both output values to be no longer than the shortest input.) In either case, the function can be written as shown below which is much more efficient than the program above since there is no longer a restriction violation which triggers code copying. The key difference is that the use of Tcotruncate makes both outputs depend on both inputs. If the inputs are known to be the same length, the use of Tcotruncate can be thought of as a declaration.

```
(defun weighted-squares* (value-vector weight-vector)
  (letS* (((values weights)
           (Tcotruncate (Evector value-vector)
                        (Evector weight-vector)))
          (squares (* values values))
          (weighted-squares (* squares weights)))
    (list (Rvector squares) (Rvector weighted-squares))))

(weighted-squares* #(1 2 3) #(2 3 4)) => (#(1 4 9) #(2 12 36))
(weighted-squares* #(1 2) #(2 3 4)) => (#(1 4) #(2 12))
(weighted-squares* #(1 2 3) #(2 3)) => (#(1 4) #(2 12))
```

Off-Line Transducers

This section and the next two describe transducers that are not on-line. Most of these functions have some inputs or outputs which are on-line. The ports which are on-line can be used freely. However, the off-line ports have to be isolated when they are used. (For ease of reference, the off-line ports all begin with the letter code "O".)

Tremove-duplicates Oitems &optional (comparator #'eql) => items

This function is analogous to remove-duplicates. It creates an OSS series that has the same elements as the off-line input Oitems with all duplicates removed. The comparator is used to determine whether or not two items are duplicates. If two items are the same, then the item which is later in the series is discarded. (As in remove-duplicates the algorithm employed is not particularly efficient, being O(n^2).) (If the Oitems input of Tremove-duplicates is such that it can be used as a destination for alterS, then the output of Tremove-duplicates can be used as a destination for alterS.)

```
(Tremove-duplicates [1 2 1 (a) (a)]) => [1 2 (a) (a)]
(Tremove-duplicates [1 2 1 (a) (a)] #'equal) => [1 2 (a)]
```

Tchunk amount Oitems => lists

This function creates an OSS series of lists of length amount of successive subseries of the off-line input Oitems. If the length of Oitems is not a multiple of amount, then the last (mod (Rlength Oitems) amount) elements of Oitems will not appear in any output chunk.

```
(Tchunk 2 [a b c d e]) => [(a b) (c d)]
(Tchunk 6 [a b c d]) => []
```

Twindow amount Oitems => lists

This function creates an OSS series of lists of length amount of subseries of the off-line input Oitems starting at each element position. If the length of Oitems is less than amount, the output will not contain any windows. The last example below shows Twindow being used to compute a moving average.

```
(Twindow 2 [a b c d]) => [(a b) (b c) (c d)]
(Twindow 4 [a b c d]) => [(a b c d)]
(Twindow 6 [a b c d]) => []
(prognS (/ (apply #'+ (Twindow 2 [2 4 6 8])) 2)) => [3 5 7]
```

Tconcatenate Oitems1 Oitems2 &rest more-Oitems => items

This function creates an OSS series by concatenating together two or more off-line input OSS series. The length of the output is the sum of the lengths of the inputs. (The elements of the individual input series are not computed until they need to be.)

```
(Tconcatenate [b c] [] [d]) => [b c d]
(Tconcatenate [] []) => []
```

TconcatenateF Enumerator Oitems => items

The Enumerator must be a quoted OSS function that is an enumerator. The function TconcatenateF applies Enumerator to each element of the off-line input Oitems and returns the series obtained by concatenating all of the results together. If Enumerator returns multiple values, then TconcatenateF will as well.

```
(TconcatenateF #'Elist [(a b) () (c d)]) => [a b c d]
(TconcatenateF #'Elist [() ()]) => []
(TconcatenateF #'Eplist [(a 1) (b 2 c 3)]) => [a b c] [1 2 3]
```

Tsubseries Oitems start &optional below => items

This function creates an OSS series containing a subseries of the elements of the off-line input Oitems from start up to, but not including, below. If below is greater than the length of Oitems, output nevertheless stops as soon as the input runs out of elements. If below is not specified, the output continues all the way to the end of Oitems. Both of the arguments start and below must be non-negative integers.

```
(Tsubseries [a b c d] 1) => [b c d]
(Tsubseries [a b c d] 1 3) => [b c]
(Rlist (Tsubseries (Elist list) 1 2)) == (subseq list 1 2)
```

If the Oitems input of Tsubseries is such that it can be used as a destination for alterS, then the output of Tsubseries can be used as a destination for alterS.

```
(let ((list '(a b c d e)))
  (alterS (Tsubseries (Elist list) 1 3) (Eup))
  list) => (a 0 1 d e)
```

The function Tsubseries terminates as soon as it has written the last output element. As a result, it is an early terminator. This can sometimes lead to conflicts with the restriction that within each on-line subexpression, there must be a data flow path from each termination point to each output. To select a subseries without using an early terminator, use Tselect, Tmask, and Eup as shown below.

```
(Tsubseries Oitems from below)
  == (Tselect (Tmask (Eup from :below below)) Oitems)
```

Tpositions Obools => indices

This function takes in an OSS series and returns an OSS series of the indexes of the non-null elements in the off-line input series.

```
(Tpositions [T nil T 44]) => [0 2 3]
(Tpositions [nil nil nil]) => []
```

```
Tmask  Omonotonic-indices => bools
```

This function is a quasi-inverse of Tpositions. The input Omonotonic-indices must be a strictly increasing OSS series of non-negative integers. The output, which is always unbounded, contains T in the positions specified by Omonotonic-indices and nil everywhere else.

```
(Tmask [0 2 3]) => [T nil T T nil nil ...]
(Tmask []) => [nil nil ...]
(Tmask (Tpositions x)) == (Tconcatenate (not (null x)) (Eoss :R nil))
```

```
Tmerge  Oitems1 Oitems2 comparator => items
```

This function is analogous to merge. The output series contains the elements of the two off-line input series. The elements of Oitems1 appear in the same order that they are read in. Similarly, the elements of Oitems2 appear in the same order that they are read in. However the elements from the two inputs are intermixed under the control of the comparator. At each step, the comparator is used to compare the current elements in the two series. If the comparator returns non-null, the current element is removed from Oitems1 and transferred to the output. Otherwise, the next output comes from Oitems2. (If, as in the first example below, the elements of the individual input series are ordered with respect to comparator, then the result will also be ordered with respect to comparator. If, as in the second example below, either input is not ordered, the result will not be ordered.)

```
(Tmerge [1 3 7 9] [4 5 8] #'<) => [1 3 4 5 7 8 9]
(Tmerge [1 7 3 9] [4 5 8] #'<) => [1 4 5 7 3 8 9]
(Tmerge x y #'(lambda (x y) T)) == (Tconcatenate x y)
```

```
Tlastp  Oitems => bools items
```

This function takes in a series and returns a series of boolean values having the same length such that the last value is T and all of the other values are nil. If the input series is unbounded, then the output series will also be unbounded and every element of the output will be nil.

It turns out that this output cannot be computed by an on-line OSS function. Therefore, if Tlastp returned only the boolean values described above, the isolation restrictions would make it impossible to use the input series and the output values together in the same computation. In order to get around this problem, Tlastp returns a second output which is identical to the input. This output can be used in lieu of the input in combination with the boolean values.

```
(Tlastp [a b c d]) => [nil nil nil T] [a b c d]
(Tlastp [a]) => [T] [a]
(Tlastp []) => [] []
(Tlastp (Eup)) => [nil nil nil ...] [0 1 2 ...]
```

As an example of using Tlastp, it is interesting to return to the example of proration discussed in conjunction with the function TscanF. Both of the proration functions presented earlier assume that the percentages always add up to 100. If this turns out not to be the case, then an exact allocation of the total is not guaranteed. The following program ensures that exact allocation will occur no matter what the percentages add up to. It does this by using Tlastp to detect which percentage is the last one.

```
(defunS Tprorate-robust (total Opercents)
    (declare (type oss Opercents))
  (letS* (((is-last percents) (Tlastp Opercents))
          (allocation (round (/ (* percents total) 100)))
          (unallocated (TscanF total #'- allocation)))
    (if is-last (Tprevious unallocated total) allocation)))

(Tprorate-robust 99 [35 45 20]) => [35 45 19]
(Tprorate-robust 99 [35 45 21]) => [35 45 19]
(Tprorate 99 [35 45 21]) => [35 45 21]
```

Selection and Expansion

Selection and its inverse are particularly important kinds of off-line transducers.

Tselect bools &optional items => Oitems

This function selects elements from a series based on a boolean series. The off-line output consists of the elements of items which correspond to non-null elements of bools. That is to say, the nth element of items is in the output iff the nth element of bools is non-null. The order of the elements in Oitems} is the same as the order of the elements in items. The output terminates as soon as either input runs out of elements. If no items input is specified, then the non-null elements of bools are themselves returned as the output of Tselect. (If the items input of Tselect is such that it can be used as a destination for alterS, then the output of Tselect can be used as a destination for alterS.)

```
(Tselect [T nil T nil] [a b c d]) => [a c]
(Tselect [a nil b nil]) => [a b]
(Tselect [nil nil] [a b]) => []
```

An interesting aspect of Tselect is that the output series is off-line rather than having the two input series be off-line. This is done in recognition of the fact that the two input series are always in synchrony with each other. Having only one port which is off-line allows more flexibility then having two ports which are off-line.

One might want to select elements out of a series based on their positions in the series rather than on boolean values. This can be done straightforwardly using Tmask as shown below.

```
(Tselect (Tmask [0 2]) [a b c d]) => [a c]
(Tselect (not (Tmask [0 2])) (Eup 10)) => [11 13 14 15 ...]
```

A final feature of Tselect in particular, and off-line ports in general, is illustrated by the program below. In this program, the Tselect causes the first Elist to get out of phase with the second Elist. As a result, it is important to think of OSS expressions as passing around series objects rather than as merely being abbreviations for loops where things are always happening in lock step. The latter point of view might lead to the idea that the output of the program below would be ((a 1) (c 2) (d 4)).

```
(letS ((tag (Elist '(a b c d e)))
       (x (Elist '(1 -2 2 4 -5))))
  (Rlist (list tag (Tselect (plusp x) x)))) => ((a 1) (b 2) (c 4))
```

TselectF pred Oitems => items

This function is the same as Tselect, except that it maps the non-OSS function pred over Oitems to obtain a series of boolean values with which to control the selection. In addition, TselectF has an off-line input rather than an off-line output (this is fractionally more efficient). The logical relationship between Tselect and TselectF is shown in the last example below.

```
(TselectF #'identity [a nil nil b nil]) => [a b]
(TselectF #'plusp [-1 2 -3 4]) => [2 4]
(TselectF pred items)
  == (letS ((var items)) (Tselect (TmapF pred var) var))
```

Texpand bools Oitems &optional (default nil) => items

This function is a quasi-inverse of Tselect. (The name is borrowed from APL.) The output contains the elements of Oitems spread out into the positions specified by the non-null elements in bools---i.e., the nth element of Oitems is in the position occupied by the nth non-null element in bools. The other positions in the output are occupied by default. The output stops as soon as bools runs out of elements, or a non-null element in bools is encountered for which there is no corresponding element in Oitems.

```
(Texpand [nil T nil T T] [a b c]) => [nil a nil b c]
(Texpand [nil T nil T T] [a]) => [nil a nil]
(Texpand [nil T] [a b c] 'z) => [z a]
(Texpand [nil T nil T T] []) => [nil]
```

Splitting

An operation which is closely related to selection, is splitting. In selection, specified elements are selected out of a series. It is not possible to apply further operations to the elements which are not selected, because they have been discarded. In contrast, splitting divides up a series into two or more parts which can be individually used. Both Tsplit and TsplitF have on-line inputs and off-line outputs. The outputs have to be off-line, because they are inherently non-synchronized with each other.

Tsplit items bools &rest more-bools => Oitems1 Oitems2 &rest more-Oitems

This function takes in a series of elements and partitions them between two or more outputs. If there are n boolean inputs then there are n+1 outputs. Each input element is placed in exactly one output series. Suppose that the nth element of bools is non-null. In this case, the nth element of items will be placed in Oitems1. On the other hand, if the nth element of bools is nil, the second boolean input (if any) is consulted in order to see whether the input element should be placed in the second output or in a later output. (As in a cond, each time a boolean element is nil, the next boolean series is consulted.) If the nth element of every boolean series is nil, then the nth element of items is placed in the last output.

```
(Tsplit [-1 -2 3 4] [T T nil nil]) => [-1 -2] [3 4]
(Tsplit [-1 -2 3 4] [T T nil nil] [nil T nil T]) => [-1 -2] [4] [3]
(Tsplit [-1 -2 3 4] [T T T T]) => [-1 -2 3 4] []
```

If the items input of Tsplit is such that it can be used as a destination for alterS, then all of the outputs of Tsplit can be used as destinations for alterS.

```
(letS* ((list '(-1 2 -3))
        (x (Elist list))
        ((x+ x-) (Tsplit x (plusp x))))
    (alterS x+ (+ x+ 10))
    (alterS x- (- x- 10))
    list) => (-11 12 -13)
```

TsplitF items pred &rest more-pred => Oitems1 Oitems2 &rest more-Oitems

This function is the same as Tsplit, except that it takes predicates as arguments rather than boolean series. The predicates must be non-OSS functions and are applied to items in order to create boolean values. The relationship between TsplitF and Tsplit is almost but not exactly as shown below.

```
(TsplitF items pred1 pred2)
 not= (letS ((var items))
        (Tsplit var (TmapF pred1 var) (TmapF pred2 var)))
```

The reason that the equivalence above does not quite hold is that, as in a cond, the predicates are not applied to individual elements of items unless the resulting value is needed in order to determine which output series the element should be placed in (e.g., if the first predicate returns non-null when given the nth element of items, the second predicate will not be called). This promotes efficiency and allows earlier predicates to act as guards for later predicates.

```
(TsplitF [-1 -2 3 4] #'minusp) => [-1 -2] [3 4]
(TsplitF [-1 -2 3 4] #'minusp #'evenp) => [-1 -2] [4] [3]
```

Reducers

Reducers produce non-OSS outputs based on OSS inputs. There are two basic kinds of reducers: ones that combine the elements of OSS series together into aggregate data structures (e.g., into a list) and ones that compute some summary value from these elements (e.g., the sum). All the predefined reducers are on-line.
A few reducers are also early terminators.
These reducers are described in the next section.

Rlist items => list

This function creates a list of the elements in items in order.

```
(Rlist [a b c]) => (a b c)
(Rlist []) => ()
(Rlist (fn (Elist x) (Elist y))) == (mapcar #'fn x y)
(Rlist (fn (Esublists x) (Esublists y))) == (maplist #'fn x y)
```

Rbag items => list

This function creates a list of the elements in items with no guarantees as to the order of the elements. The function Rbag is more efficient than Rlist.

```
(Rbag [a b c]) => (c a b) ;in some order
(Rbag []) => ()
```

Rappend lists => list

This function creates a list by appending the elements of lists together in order.

```
(Rappend [(a b) nil (c d)]) => (a b c d)
(Rappend []) => ()
```

Rnconc lists => list

This function creates a list by nconcing the elements of lists together in order. The function Rnconc is faster than Rappend, but modifies the lists in the OSS series lists.

```
(Rnconc [(a b) nil (c d)]) => (a b c d)
(Rnconc []) => ()
(let ((x '(a b))) (Rnconc (Eoss x x))) => (a b a b a b ...)
(Rnconc (fn (Elist x) (Elist y))) == (mapcan #'fn x y)
(Rnconc (fn (Esublists x) (Esublists y))) == (mapcon #'fn x y)
```

Ralist keys values => alist

This function creates an alist containing keys and values. It terminates as soon as either of the inputs runs out of elements. If there are duplicate keys, they will be put on the alist, but order is preserved.

```
(Ralist [a b] [1 2]) => ((a . 1) (b . 2))
(Ralist [a b] []) => ()
(Ralist keys values) == (Rlist (cons keys values))
```

Rplist indicators values => plist

This function creates a plist containing keys and values. It terminates as soon as either of the inputs runs out of elements. If there are duplicate indicators, they will be put on the plist, but order is preserved.

```
(Rplist [a b a] [1 2 3]) => (a 1 b 2 a 3)
(Rplist [a b] []) => ()
(Rplist keys values) == (Rnconc (list keys values))
```

Rhash keys values &rest option-plist => table

This function creates a hash table containing keys and values. It terminates as soon as either of the inputs runs out of elements. The option-plist can contain any options acceptable to make-hash-table. The option-plist cannot refer to variables bound by letS.

```
(Rhash [color name] [brown fred]) => #<hash-table 23764432>
;;hash table containing color->brown, name->fred

(Rhash [color name] []) => #<hash-table 23764464>
;;empty hash table
```

Rvector items &key :size &rest option-plist => vector

This function creates a vector containing the elements of items in order. The option-plist can contain any options acceptable to make-array. The option-plist cannot refer to variables bound by letS.

The function Rvector operates in one of two ways. If the :size argument is supplied, then Rvector assumes that items will contain exactly :size elements. A vector is created of length :size with the options specified in option-plist and the elements of items are stored in it. (If items has fewer than :size elements, some of the slots in the vector will be left in their initial state. If items has more than :size elements, an error will ensue.) In this mode, Rvector is very efficient, but rather inflexible.

```
(Rvector [1 2 3] :size 3) => #(1 2 3)
(Rvector [#\B #\A #\R] :size 3 :element-type 'string-char) => "BAR"
(Rvector [1] :size 4 :initial-element 0) => #(1 0 0 0)
```

If the :size argument is not supplied, then Rvector allows for the creation of an arbitrarily large vector. It does this by using vector-push-extend. In order for this to work, it forces :adjustable to be T and :fill-pointer to be 0 no matter what is specified in the options-list. In this mode, an arbitrary number of input elements can be handled, however, things are much less efficient, since the vector created is not a simple vector.

```
(Rvector [1 2 3]) => #(1 2 3)
(Rvector []) => #()
(Rvector [#\B #\A #\R] :element-type 'string-char) => "BAR"
```

To store a series in a preexisting vector, use alterS of Evector.

```
(let ((v '#(a b c)))
  (alterS (Evector v) (Eoss 1 2))
  v) => #(1 2 c)
```

Rfile name items &rest option-plist => T

This function creates a file named name and writes the elements of items into it using print. The option-plist can contain any of the options accepted by open except :direction which is forced to be :output. All of the ordinary printer control variables are obeyed during the printout. The value T is always returned. The option-plist cannot refer to variables bound by letS.

```
(Rfile "test.lisp" ['(a) '(1 2) T] :if-exists :append) => T
;;The output "
;;(a)
;;(1 2)
;;T " is printed into the file "test.lisp".
```

Rlast items &optional (default nil) => item

This function returns the last element of items. If items is of zero length, default is returned.

```
(Rlast [a b c]) => c
(Rlast [] 'z) => z
```

Rlength items => number

This function returns the number of elements in items.

```
(Rlength [a b c]) => 3
(Rlength []) => 0
```

Rsum numbers => number

This function computes the sum of the elements in numbers. These elements must be numbers, but they need not be integers.

```
(Rsum [1 2 3]) => 6
```

```
(Rsum []) => 0
(Rsum [1.1 1.2 1.3]) => 3.6
```

Rmax numbers => number

This function computes the maximum of the elements in numbers.
These elements must be non-complex numbers, but they need not be integers.
The value nil is returned if numbers has length zero.

```
(Rmax [2 1 4 3]) => 4
(Rmax []) => nil
(Rmax [1.2 1.1 1.4 1.3]) => 1.4
```

Rmin numbers => number

This function computes the minimum of the elements in numbers.
These elements must be non-complex numbers, but they need not be integers.
The value nil is returned if numbers has length zero.

```
(Rmin [2 1 4 3]) => 1
(Rmin []) => nil
(Rmin [1.2 1.1 1.4 1.3]) => 1.1
```

ReduceF init function items => result

This function is analogous to reduce. In addition, it is similar to TscanF except that init is not optional and the final value of the accumulator is the only value returned as shown in the last example below. If items is of length zero, init is returned. As with TscanF, function must be a non-OSS function and the value of init is typically chosen to be a left identity of function. It is important to remember that the elements of items are used as the second argument of function. The order of arguments is chosen to highlight this fact.

```
(ReduceF 0 #'+ [1 2 3]) => 6
(ReduceF 0 #'+ []) => 0
(ReduceF 0 #'+ x) == (Rsum x)
(ReduceF init function items)
  == (letS ((var init))
       (Rlast (TscanF var function items) var))
```

In order to do reduction without an initial seed value, use Rlast of TscanF. Note that although a seed value does not have to be specified, a value to be returned if there are no elements in items still has to be specified.

```
(Rlast (TscanF #'max x) nil) == (Rmax x)
```

Early Reducers

The following four reducers are early terminators. Each of these functions has a non-early variant denoted by the suffix "-late". The early variants are more efficient, because they terminate as soon as they have determined a result.
This may be long before any of the input series run out of elements. However, as discussed at the end of this section, one has to be somewhat careful when using an early reducer in an OSS expression.

Rfirst items &optional (default nil) => item

Rfirst-late items &optional (default nil) => item

Both of these functions return the first element of items. If items is of zero length, default is returned. The only difference between the functions is that Rfirst stops immediately after reading the first element of items, while Rfirst-late does not terminate until items runs out of elements.

```
(Rfirst [a b c]) => a
(Rfirst [] 'z) => z
```

Rnth n items &optional (default nil) => item

Rnth-late n items &optional (default nil) => item

Both of these functions return the nth element of items. If n is greater than or equal to the length of items, default is returned. The only difference between the functions is that Rnth stops immediately after reading the nth element of items, while Rnth-late does not terminate until items runs out of elements.

```
(Rnth 1 [a b c]) => b
(Rnth 1 [] 'z) => z
```

Rand bools => bool

Rand-late bools => bool

Both of these functions compute the and of the elements in bools. As with the function and, nil is returned if any element of bools is nil. Otherwise the last element of bools is returned. The value T is returned if bools has length zero. The only difference between the functions is that Rand terminates as soon as a nil is encountered in the input, while Rand-late does not terminate until bools runs out of elements.

```
(Rand [a b c]) => c
(Rand [a nil c]) => nil
(Rand []) => T
(Rand (pred (Esequence x) (Esequence y))) == (every #'pred x y)
```

Ror bools => bool

Ror-late bools => bool

Both of these functions compute the or of the elements in bools. As with the function or, nil is returned if every element of bools is nil. Otherwise the first non-null element of bools is returned. The value nil is returned if bools has length zero. The only difference between the functions is that Ror terminates as soon as a non-null value is encountered in the input, while Ror-late does not terminate until bools runs out of elements.

```
(Ror [a b c]) => a
(Ror [a nil c]) => a
(Ror []) => nil
(Ror (pred (Esequence x) (Esequence y))) == (some #'pred x y)
```

Care must be taken when using early reducers. As discussed in the section on restrictions, OSS expressions are required to obey the restriction that within each on-line subexpression, there must be a data flow path from each termination point to each output. Early reducers interact with this restriction since early reducers are termination points. As a result, there must be a data flow path from each early reducer to each output of the containing on-line subexpression.

Since reducers compute non-OSS values, they directly compute outputs of on-line subexpressions. As a result, it is impossible for there to be a data flow path from a reducer to any output other than the output the reducer itself computes. Therefore, the use of an early reducer will trigger code copying unless that reducer computes the only output of the on-line subexpression.

For example, consider the following four expressions. The first two expressions return the same result. However, the first is more efficient. This is a prototypical example of a situation where it is better to use an early reducer. In contrast, although the last two expressions also return the same results, the second of the expressions is more efficient. The problem is that in the first of these expressions, there is no data flow path from the use of Rfirst to the second output. In order to fix this problem the OSS macro package duplicates the list enumeration. It is more efficient to use a non-early reducer as in the last example.

```
(letS ((x (Elist '(1 2 -3 4 5 -6 -7 8))))
  (Rfirst (TselectF #'minusp x))) => -3

(letS ((x (Elist '(1 2 -3 4 5 -6 -7 8))))
  (Rfirst-late (TselectF #'minusp x))) => -3

(letS ((x (Elist '(1 2 -3 4 5 -6 -7 8))))   ;Signals warning 18
  (valS (Rfirst (TselectF #'minusp x))
      (Rsum x))) => -3 4

(letS ((x (Elist '(1 2 -3 4 5 -6 -7 8))))
  (valS (Rfirst-late (TselectF #'minusp x))
      (Rsum x))) => -3 4
```

Series Variables

The principal way to create OSS variables is to use the form letS. (These variables are also created by the forms lambdaS and defunS.)

letS var-value-pair-list {decl}* &body expr-list => result

The form letS is syntactically analogous to let. Just as in a let, the first subform is a list of variable-value pairs. The letS form defines the scope of these variables and gives them the indicated values. As in a let, one or more declarations can follow the variable-value pairs. These can be used to specify the types of the variables.

The variables created by letS can be OSS variables or non-OSS variables. Which are which is determined by the type of the value that is bound to the variable. As in let, the variables are bound in parallel. In the example below, y is an OSS variable while x and z are non-OSS variables.

```
(letS ((x '(1 2 3))
       (y (Elist '(1 2 3)))
       (z (Rsum (Elist '(1 2 3)))))
  (list x (Rmax y) z)) => ((1 2 3) 3 6)
```

Unlike let, letS does not support degenerate variable-value pairs which consist solely of a variable. (Since letS variables cannot be assigned to, see below, degenerate pairs would be of little value.)

```
(letS (x) ...)                              ;Signals error 9
```

The following example illustrates the use of a declaration in a letS. Declarations are handled in the same way that they are handled in a let.

```
(letS ((x (Elist '(1 2 3))))
    (declare (type integer x))
  (Rsum x)) => 6
```

The form letS goes beyond let to include the functionality of multiple-value-bind. A variable in a variable-value pair can be a list of variables instead of a single variable. When this is the case, the variables pick up the first, second, etc. results returned by the value expression. (If there is only one variable, it gets the first value. If nil is used in lieu of
a variable, the corresponding value is ignored.) If there are fewer variables than values, the extra values are ignored. Unlike multiple-value-bind, letS signals an error if there are more variables than values. (Note that there is no form multiple-value-bindS and that the form multiple-value-bind cannot be used inside of an OSS expression to bind the results of an OSS function.)

```
(letS (((key value) (Ealist '((a . 1) (b . 2)))))
  (Rlist (list key value))) => ((a 1) (b 2))

(letS ((key (Ealist '((a . 1) (b . 2)))))
  (Rlist key)) => (a b)

(letS (((nil value) (Ealist '((a . 1) (b . 2)))))
  (Rlist value)) => (1 2)

(letS (((key value x) (Ealist '((a . 1) (b . 2)))))
  (Rlist (list key value x)))           ;Signals error 8
```

The expr-list of a letS has the effect of grouping several OSS expressions together. The value of the last form in the expr-list is returned as the value of the letS. This value may be an OSS value or a non-OSS value.

In addition to placing all of the expressions in the same letS binding scope, the grouping imposed by the expr-list causes the entire body to become an OSS expression. This can alter the way implicit mapping is applied by including non-OSS functions in the OSS expression.

The restricted nature of OSS variables.
There are a number of ways in which the variables bound by letS (or lambdaS and defunS) are more restricted than the ones bound by let. For the most part, these restrictions stem from the fact that when the OSS macro package transforms an OSS expression into a loop, it rearranges the expressions extensively. This forces letS variable scopes to be supported by variable renaming rather than binding. One result of this is that it is not possible to declare (or proclaim) a letS variable to be special.
(Standard Common Lisp does not provide any method for determining whether or not a variable has been proclaimed special. As a result, the OSS macro package is unable to issue an error message when a special letS variable is encountered. The Symbolics Common Lisp version of the OSS macro package does issue an error message.)

```
(proclaim '(special z))
(letS ((z (Elist '(1 2 3)))) (Rsum z)) ;erroneous expression
```

Another limitation is that programmers are not allowed to assign values to letS variables in the body of a letS. (This restriction applies whether or not the variables contain OSS values.) The only time letS variables can be given a value is the moment they are bound. (Although assignment could be supported easily enough, the rearrangements introduced by the OSS macro package would make it very confusing for a programmer to figure out exactly what would happen in a given situation. In particular, naively applying implicit mapping to setq would lead to peculiar results. In addition, outlawing assignments enhances the functional nature of the OSS macro package.) An error message is issued whenever such an assignment is attempted.

```
(lets ((x (Elist '(1 2 3))))
  (setq x (1+ x))                    ;Signals error 12
  (Rlist x))
```

Another aspect of letS variables is that their scope is somewhat limited. In particular, letS variables can be referenced in a letS or mapS which is inside the letS which binds them. However, they cannot be referenced in lambda or lambdaS. (As above, this limitation is imposed in order to avoid confusions due to rearrangements. Further, it is not obvious what it would mean to refer to an OSS variable in a lambda. Should some sort of implicit mapping be applied?) No attempt is made to issue error messages in this situation. Rather, the variable reference in question is merely treated as a free variable.

```
(let ((x 4))
  (letS ((x (Elist '(1 2 3))))
    (Rlist (TmapF #'(lambda (y) (+ x y)) x)))) => (5 6 7)
```

letS* var-value-pair-list {decl}* &body expr-list => result

The form letS* is exactly the same as letS except that the variables are bound sequentially instead of in parallel.

```
(letS* ((x '(1 2 3))
        (y (Elist x))
        (z (Rsum y)))
  (list x (Rmax y) z)) => ((1 2 3) 3 6)
```

prognS &body expr-list => result

As shown below, prognS is identical to letS except that it cannot contain any variable-value pairs or declarations. It is a degenerate form whose only function is to delineate an OSS expression. This can alter the way implicit mapping is applied by including non-OSS functions in the OSS expression.

```
(prognS . expr-list) == (letS () . expr-list)
```

Complete OSS expressions do not return OSS values.
A key point relevant to the discussion above is that syntactically complete OSS expressions are not allowed to return OSS values. This is relevant, because letS and prognS are often used in such a way that an OSS series gratuitously ends up as the return value. For example, the main intent of the expression below is to print out the elements of the list. However, as written, the expression appears to return an OSS series of the values produced by prin1. Because expressions like the one below are relatively common, it was decided not to issue an error message in this situation. Rather, the OSS value is simply discarded and no value is returned.

```
(prognS (prin1 (Elist '(1 2)))) =>
;;The output "12" is printed.
```

It might be the case that the programmer actually desires to have a physical series returned in the example above. This can be done by using a reducer such as Rlist or Rvector as shown below.

```
(prognS (Rlist (prin1 (Elist '(1 2))))) => (1 2)
;;The output "12" is printed.
```

Preventing complete OSS expressions from returning OSS values does not limit what can be written, because programmers can always return a non-OSS series. This can be a bit cumbersome at times, but it is highly preferable to the large inefficiencies which would be introduced by automatically constructing physical representations for OSS series in situations where the returned values are not used in further computation.

Coercion of Non-Series to Series

If an OSS input of an OSS function is applied to a non-series value, the type conflict is resolved by converting the non-OSS value into a series by inserting Eoss. That is to say, a non-OSS value acts the same as an unbounded OSS series of the value.

```
(Ralist (Elist '(a b)) (* 2 3))
  == (Ralist (Elist '(a b)) (Eoss :R (* 2 3))) => ((a . 6) (b . 6))
```

Using Eoss to coerce a non-OSS value to an OSS series has the effect of only evaluating the expression which computes the value once. This has many advantages with regard to efficiency, but may not always be what is desired. Multiple evaluation can be specified by using TmapF or mapS.

```
(Ralist (Elist '(a b)) (gensym)) => ((a . #:G004) (b . #:G004))
(Ralist (Elist '(a b)) (TmapF #'gensym)) => ((a . #:G004) (b . #:G005))
```

Implicit Mapping

Mapping operations can be created by using TmapF. However, in the interest of convenience, two other ways of creating mapping operations are supported. The most prominent of these is implicit mapping. If a non-OSS function appears in an OSS expression and is applied to one or more arguments which are OSS series, the type conflict is resolved by automatically mapping the function over these series.

```
(Rsum (car (Elist '((1) (2)))))
  == (Rsum (TmapF #'car (Elist '((1) (2))))) => 3

(Rsum (* 2 (Elist '(1 2))))
  == (Rsum (TmapF #'(lambda (x) (* 2 x)) (Elist '(1 2)))) => 6
```

As shown in the second example, implicit mapping actually applies to entire non-OSS subexpressions rather than merely to individual functions. This promotes efficiency and makes sure that related groups of functions are mapped together. However, it is not always what is desired. For instance, in the first example below, the call on gensym gets mapped in conjunction with the call on list. This causes each list to contain a separate gensym variable. It might be the case that the programmer wants to have the same gensym variable in each list. This can be achieved by inserting an Eoss as shown in the second example. (Inserting a Eoss here and there can promote efficiency by avoiding unnecessary recomputation.)

```
(Rlist (list (Elist '(a b)) (gensym)))
  == (Rlist (TmapF #'(lambda (x) (list x (gensym)))
                (Elist '(a b)))) => ((a #:G002) (b #:G003))

(Rlist (list (Elist '(a b)) (Eoss :R (gensym))))
  == (Rlist (TmapF #'list
                (Elist '(a b))
                (Eoss :R (gensym)))) => ((a #:G002) (b #:G002))
```

In order to be implicitly mapped, a non-OSS function must appear inside of an OSS expression. For example, the instance of prin1 in the first example below does not get implicitly mapped, because it is not in an OSS expression. Implicit mapping of the prin1 can be forced by using prognS as shown in the second example above.

```
(prin1 (Elist '(1 2))) => nil
;;The output "NIL" is printed.

(prognS (prin1 (Elist '(1 2)))) =>
;;The output "12" is printed.
```

(The result of the first example above is that NIL gets printed. This happens because (Elist '(1 2 3)) is a syntactically complete OSS expression and is therefore not allowed to return a series. It returns no values instead. The function prin1 demands a value anyway, and gets nil.)

Another aspect of implicit mapping is that a non-OSS function will not be mapped unless it is applied to a series. This is usually, but not always, what is desired. Consider the first expression below. The instance of prin1 is mapped over x. However, the instance of princ is not applied to a series and is therefore not mapped. If the programmer intends to print a dash after each number, he has to do something in order to get the princ to be mapped. This could be done using TmapF or mapS. However, the best thing to do is to group the two printing statements into a single subexpression as shown in either of the last two examples below. This grouping shows the relationship between the printing operations and causes them to be mapped together.

```
(letS ((x (Elist '(1 2 3))))
  (prin1 x)
  (princ "-")) => "-"
;;The output "123-" is printed.

(letS ((x (Elist '(1 2 3))))
  (progn (prin1 x) (princ "-"))) =>
;;The output "1-2-3-" is printed.

(letS ((x (Elist '(1 2 3))))
  (format T "~A-" x)) =>
;;The output "1-2-3-" is printed
```

Ugly details.
Implicit mapping is easy to understand when applied in simple situations such as the ones above. However, it can be applied to any Lisp form. Things become somewhat more complicated when control constructs (e.g., if) and binding constructs (e.g., let) are encountered. The example below shows the implicit mapping of an if. This creates a lambda expression containing a conditional which is mapped over a series. A key thing to notice in this example is that implicit mapping of if is very different

from a use of Tselect. In particular, the mapped if returns a value corresponding to every input, while the Tselect does not.

```
(Rlist (if (plusp (Elist '(10 -11 12))) (Eup)))
  == (Rlist (TmapF #'(lambda (x y) (if (plusp x) y))
                   (Elist '(10 -11 12)) (Eup))) => (0 nil 2)

(Rlist (Tselect (plusp (Elist '(10 -11 12))) (Eup))) => (0 2)
```

Another aspect of the way conditionals are handled inside of an OSS expression is illustrated below. When an OSS expression is being processed in order to determine what should be implicitly mapped, the expression is broken up into OSS pieces and non-OSS pieces. If the argument of a conditional is an OSS expression, this argument will end up in a separate piece from the conditional itself. One result of this is that the argument will always be evaluated and the conditional will therefore lose its power to control when the argument should be evaluated. This effect will happen even if, as in the example below, the conditional does not have to be mapped. The three examples below all produce the same value, but the first two always evaluate (Rlist (abs (Elist x))) while the last may not.

```
(prognS (if (Ror (minusp (Elist x)))
            (Rlist (abs (Elist x)))
            x))
  == (prognS (funcall #'(lambda (y z) (if y z x))
                      (Ror (minusp (Elist x)))
                      (Rlist (abs (Elist x)))))
 not= (if (Ror (minusp (Elist x)))
          (Rlist (abs (Elist x)))
          x)
```

The following example shows the implicit mapping of a let. (Among other things, this illustrates that such expressions are far from clear. In general it is better to use letS as in the second example.)

```
(Rlist (let ((double (* 2 (Elist '(1 2))))) (* double double)))
  == (Rlist (TmapF #'(lambda (x)
                       (let ((double (* 2 x))) (* double double)))
                   (Elist '(1 2)))) => (4 16)

(letS ((double (* 2 (Elist '(1 2)))))
  (Rlist (* double double))) => (4 16)
```

A problem with the implicit mapping of a let (or other binding forms) is that the implicit mapping transformation potentially moves subexpressions out of the scope of the binding form in question. This can change the meaning of the expression if any of these subexpressions contain an instance of a variable bound by the binding form. For instance, in the example above, the transformation moves the subexpression (Elist '(1 2)) out of the scope of the let. This would cause a problem if this subexpression referred to the variable double.

In recognition of this problem, a warning message is issued whenever implicit mapping of a binding form causes a variable reference to move out of a form that binds it. Whenever it occurs, this problem can be alleviated by using letS as shown above.

A final complexity involves forms like return, return-from, throw, etc. These forms are implicitly mapped like any other non-OSS form. When they get evaluated, they will cause an exit. However, the loop produced by the OSS macro does not contain a boundary which is recognized by any of these forms (e.g., it does not create a prog or catch). As a result, such a boundary must be defined which will serve as the reference point. Needless to say, the final results of the OSS expression will not be computed if the expression is exited in this way.

Nested loops.
Implicit mapping is applied when non-OSS functions receive OSS values. However, implicit mapping is not applied when OSS functions receive OSS values, even if these values are passed to non-OSS inputs. As illustrated below, whenever this situation occurs, an error message is issued.

```
(Elist (Elist '((1 2) (3 4))))                ;Signals error 14
```

There are situations corresponding to nested loops where it would be reasonable to implicitly map subexpressions containing OSS functions. For example, one might write the following expression in order to copy a list of lists.

```
(Rlist (Rlist (Elist (Elist '((1 2) (3 4)))))) ;Signals error 14
(Rlist (TmapF #'(lambda (x) (Rlist (Elist x)))
          (Elist '((1 2) (3 4))))) => ((1 2) (3 4))
```

Nevertheless, expressions like the first one above are forbidden. This is done for two reasons. First, in more complex situations OSS expressions corresponding to nested loops become so confusing that such expressions are very hard to understand. As a result, they are not very useful. Second, experience suggests that a large proportion of situations where mapping of OSS functions might be done arise from programming errors rather than an intention to have a nested loop. Outlawing these expressions makes it possible to find these errors more quickly.

(The following example shows that there is no problem with having one loop computation following another. There are no type conflicts in this situation and no implicit mapping is required.)

```
(Rsum (Evector (Rvector (Elist '(1 2))))) => 3
```

Needless to say, it would be unreasonable if there were no way to write OSS expressions corresponding to nested loops. First of all, this can always be done using TmapF as shown above. However, this can be rather cumbersome. To alleviate this difficulty, an additional form (mapS) is introduced which facilitates the expression of nested computations.

mapS &body expr-list => items

The expr-list consists of one or more expressions. These expressions are treated as the body of a function and mapped over any free OSS variables which appear in them. That is to say, the first element of the output is computed by evaluating the expressions in an environment where each OSS variable is bound to the first element of the corresponding series. The second element of the output is computed by evaluating the expressions in an environment where each OSS variable is bound to the second element of the corresponding series, etc. The way mapS could be used to copy a list-of-lists is shown below. A letS has to be used, because mapS requires that the series being mapped over must be held in a variable.

```
(letS ((z (Elist '((1 2) (3 4)))))
  (Rlist (mapS (Rlist (Elist z)))))
```

```
    == (letS ((z (Elist '((1 2) (3 4)))))
         (Rlist (TmapF #'(lambda (x)
                          (Rlist (Elist x))) z))) => ((1 2) (3 4))

  (Rlist
    (mapS
      (Rlist (Elist (Elist '((1 2) (3 4))))))) ;Signals error 14
```

Implicit mapping is very valuable.
From the above, it can be seen that although implicit mapping is simple in simple situations, there are a number of situations where it becomes quite complex. There is no question that these complexities dilute the value of implicit mapping. Nevertheless, experience suggests that implicit mapping is so valuable that, warts and all, it is perhaps the most useful single feature of OSS expressions.

Literal Series Functions

Just as it is very convenient to be able to specify a literal non-OSS function using lambda, it is sometimes convenient to be able to specify a literal OSS function.

lambdaS var-list {decl}* &body expr-list

The form lambdaS is analogous to lambda except that some of the arguments can have OSS series passed to them and the return value can be an OSS series. The var-list is simpler than the lambda lists which are supported by lambda. In particular, the var-list must consist solely of variable names. It cannot contain any of the lambda list keywords such as &optional and &rest. As in a letS, the variables in the var-list cannot be assigned to in the expr-list or referenced inside of a nested lambda or lambdaS.

As in a lambda, the body can begin with one or more declarations. All of the arguments which are to receive OSS values have to be declared inside the lambdaS using the declaration type oss (see below). All of the other arguments are assumed to correspond to non-OSS values. Just as in a letS, the declarations may contain other kinds of declarations besides type oss declarations. However, the variables in the var-list cannot be declared (or proclaimed) to be special.

The expr-list is a list of expressions which are grouped together into an OSS expression as in a letS or prognS. The value of the function specified by a lambdaS is the value of the last form in the expr-list. This value may or may not be an OSS series.

In many ways, lambdaS bears the same relationship to letS that lambda bears to let. However, there is one key difference. The expr-list in a lambdaS cannot refer to any free variables which are bound by a letS, defunS, or another lambdaS. Each lambdaS is processed in complete isolation from the OSS expression which surrounds it. The only values which can enter or leave a lambdaS are specified by the var-list and non-OSS variables which are bound outside of the entire containing OSS expression.

Another key feature of lambdaS is that the only place where it can validly appear is as the quoted first argument of funcallS (see below), or as an argument to a macro which will eventually expand in such a way that the lambdaS will end up as the quoted first argument of a funcallS.

The following example illustrates the use of lambdaS. It shows an anonymous OSS function identical to Rsum.

```
(funcallS #'(lambdaS (x)
              (declare (type oss x))
            (ReduceF 0 #'+ x))
          (Elist '(1 2 3))) => 6
```

type oss &rest variable-list

This type declaration can only be used inside of a declare inside of a lambdaS or a defunS. It specifies that the variables carry OSS values.

funcallS function &rest expr-list => result

This is analogous to funcall except that function can be an OSS function. In particular, it can be the quoted name of a series function, a quoted lambdaS, or a macro call which expands into either of the above. It is also possible for function to be a non-OSS function, in which case funcallS is identical to TmapF. If function is an expression which evaluates to a function (as opposed to a literal function), then it is assumed to be a non-OSS function.

```
(funcallS #'Elist '(1 2)) == (Elist '(1 2)) => [1 2]
(funcallS #'(lambdaS (y) (declare (type oss y)) (* 2 y))
          (Elist '(1 2))) => [2 4]
(funcallS #'car [(1) (2)]) => [1 2]
(funcallS #'car '(1 2)) => [1 1 1 1 ...]
```

The number of expressions in expr-list must be exactly the same as the number of arguments expected by function. If not, an error message is issued. In addition, the types of values (either OSS series or not) returned by the expressions should be the same as the types which are expected by function. If not, coercion of non-series to series will be applied if possible in order to resolve the conflict.

Defining Series Functions

An important aspect of the OSS macro package is that it makes it easy for programmers to define new OSS functions. Straightforward OSS functions can be defined using the facilities outlined below. More complex OSS functions can be defined using the subprimitive facilities described in [6].

defunS name lambda-list {doc} {decl}* &body expr-list

This is analogous to defun, but for OSS functions. At a simple level, defunS is just syntactic sugar which defines a macro that creates a funcallS of a lambdaS. The lambda-list, declarations, and expression list are restricted in exactly the same way as in a lambdaS except that the standard lambda list keywords &optional and &key are allowed in the lambda-list.

```
(defunS Rlast (items &optional (default nil))
    "Returns the last element of an OSS series"
    (declare (type oss items))
  (ReduceF default #'(lambda (state x) x) items))
  == (defmacro Rlast (items &optional (default 'nil))
      "Returns the last element of an OSS series"
```

```
`(funcallS #'(lambdaS (items default)
               (declare (type oss items))
             (ReduceF default #'(lambda (state x) x) items))
           ,items ,default))
```

However, at a deeper level, there is a key additional aspect to defunS. Preprocessing and checking of the resulting lambdaS is performed when the defunS is evaluated (or compiled), rather than when the resulting OSS function is used. This saves time when the function is used. More importantly, it leads to better error messages because error messages can be issued when the defunS is initially encountered, rather than when the OSS function defined is used.

Although the lambda list keywords &optional and &key are supported by defunS, it should be realized that they are supported in the way they are supported by macros, not the way they are supported by functions. In particular, when keywords are used in a call on the OSS function being defined, they have to be literal keywords rather than computed by an expression. In addition, initialization forms cannot refer to the run-time values of other arguments, because these are not available at macro-expansion-time. They are also not allowed to refer to the macro-expansion-time values of the other arguments. They must stand by themselves when computing a value. A quote is inserted so that this value will be computed at run-time rather than at macro-expansion-time. (In the example above, (default nil) becomes (default 'nil).)

It may seem unduly restrictive that defunS does not support all of the standard keywords in lambda-list. However, this is not that much of a problem because defmacro can be used directly in situations where these capabilities are desired. For example, Tconcatenate is defined in terms of a more primitive OSS function Tconcatenate2 as follows.

```
(defmacro Tconcatenate (Oitems1 Oitems2 &rest more-Oitems)
  (if (null more-Oitems)
    `(Tconcatenate2 ,Oitems1 ,Oitems2)
    `(Tconcatenate2 ,Oitems1 (Tconcatenate ,Oitems2 .,more-Oitems))))
```

Using defmacro directly also makes it possible to define new higher-order OSS functions. For example, an OSS function analogous to substitute-if could be defined as follows. (The Eoss ensures that newitem will only be evaluated once.)

```
(defmacro Osubstitute-if (newitem test items)
  (let ((var (gensym)))
    `(letS ((,var ,items))
      (if (funcall ,test ,var) (Eoss :R ,newitem) ,var))))

(Osubstitute-if 3 #'minusp [1 -1 2 -3]) => [1 3 2 3]
```

Multiple Values

The OSS macro package supports multiple values in a number of contexts. As discussed above, letS can be used to bind variables to multiple values returned by an OSS function. Faculties are also provided for defining OSS functions which return multiple values. The support for multiple values is complicated by the fact that the OSS macro package implements all communication of values by using variables. As a result, it is not possible to support the standard Common Lisp feature that multiple values can coexist with single values without the programmer having to pay much attention to what is going on. When using OSS expressions, the programmer has to be explicit about how many values are being passed around.

valS &rest expr-list => &rest multiple-value-result

This is analogous to values except that it can operate on OSS values. It takes in the values returned by n different expressions and returns them as n multiple values. It enforces the restriction that the values must either all be OSS values or all be non-OSS values. The following example shows how a simple version of Eplist could be defined.

```
(defunS simple-Eplist (place)
  (letS ((plist (EnumerateF place #'cddr #'null)))
    (valS (car plist) (cadr plist))))
```

It is possible to use values in an OSS expression. However, the results will be very different from the results obtained from using valS. The values will be implicitly mapped like any other non-OSS form. The value ultimately returned will be the single value returned by TmapF.

```
(prognS (valS (Elist '(1 2)) (Elist '(3 4)))) => [1 2] [3 4]

(prognS (values (Elist '(1 2)) (Elist '(3 4))))
  == (prognS (TmapF #'(lambda (x y) (values x y))
                    (Elist '(1 2)) (Elist '(3 4)))) => [1 2]
```

pass-valS n expr => &rest multiple-value-result

This function is used essentially as a declaration. It tells the OSS macro package that the form expr returns n multiple values which the programmer wishes to have preserved in the context of the OSS expression. (This is needed, because Common Lisp does not provide any compile-time way to determine the number of arguments that a function will return.) The first example below enumerates a list of symbols and returns a list of the internal symbols, if any, which correspond to them. The second example defines a two valued OSS function which locates symbols.

```
(letS* ((names (Elist '(zots Elist zorch)))
        ((symbols statuses) (pass-valS 2 (find-symbol (string names))))
        (internal-symbols (Tselect (eq statuses :internal) symbols)))
  (Rlist internal-symbols)) => (zots zorch)

(defunS find-symbols (names)
    (declare (type oss names))
  (pass-valS 2 (find-symbol (string names))))

(find-symbols [zots Elist zorch])
=> [zots Elist zorch] [:internal :inherited :internal]
```

The form pass-valS never has to be used in conjunction with an OSS function, because the OSS macro package knows how many values every OSS function returns. Similarly, pass-valS never has to be used when multiple values are being bound by letS, because the syntax of the letS indicates how many values are returned. (As a result, the pass-valS in the first example above is not necessary.) However, in situations such as the second example above, pass-valS must be used.

Alteration of Values

The transformations introduced by the OSS macro package are inherently antagonistic to the transformations introduced by the macro setf. In particular, OSS function calls cannot be used as the destination of a setf. In order to get around this problem, the OSS macro package supports a separate construct which is in fact more powerful than setf.

```
alterS  destinations items => items
```

This form takes in a series of destinations and a series of items and stores the items in the destinations. It returns the series of items. Like setf, alterS cannot be applied to a destination unless there is an associated definition for what should be done (see the discussion of alterableS in [6]). The outputs of the predefined functions Elist, Ealist, Eplist, Efringe, Evector, and Esequence are alterable. The effects of this alteration are illustrated in conjunction with the descriptions of these functions. For example, the following sets all of the elements in a list to nil.

```
(let ((list '((a . 1) (b . 2) (c . 3))))
  (alterS (Elist list) nil)
  list) => (nil nil nil)
```

As a related example, consider the following. Although setf cannot be applied to an OSS function, it can be applied to a non-OSS function in an OSS expression. In the example below, setf is used to set the cdr of each element of a list to nil.

```
(let ((list '((a . 1) (b . 2) (c . 3))))
  (prognS (setf (cdr (Elist list)) nil))
  list) => ((a) (b) (c))
```

A key feature of alterS is that (in contrast to setf) a structure can be altered by applying alterS to a variable which contains enumerated elements of the structure. This is useful because the old value in a structure can be used to decide what new value should be put in the structure. (When alterS is applied to such a variable it modifies the structure being enumerated but does not change the value of the variable.)

```
(letS* ((v '#(1 2 3))
        (x (Evector v)))
  (alterS x (* x x))
  (valS (Rlist x) v)) => (1 2 3) #(1 4 9)
```

Another interesting aspect of alterS is that it can be applied to the outputs of a number of transducers. This is possible whenever a transducer passes through unchanged a series of values taken from an input which is itself alterable. This can happen with the transducers Tuntil, TuntilF, Tcotruncate, Tremove-duplicates, Tsubseries, Tselect, TselectF, Tsplit, and TsplitF. For example, the following takes the absolute value of the elements of a vector.

```
(letS* ((v '#(1 -2 3))
        (x (TselectF #'minusp (Evector v))))
  (alterS x (- x))
  v) => #(1 2 3)
```

Debugging

The OSS macro package supports a number of features which are intended to facilitate debugging. One example of this is the fact that the macro package tries to use the variable names which are bound by a letS in the code produced. Since the macro package is forced to use variable renaming in order to implement variable scoping, it cannot guarantee that these variable names will be used. However, there is a high probability that they will. If a break occurs in the middle of an OSS expression, these variables can be inspected in order to determine what is going on. If a letS variable holds an OSS series, then the variable will contain the current element of the series. For example, the OSS expression below is transformed into the loop shown. (For a discussion of how this transformation is performed see [6].)

```
(letS* ((v (get-vector user))
        (x (Evector v)))
  (Rsum x))

(let (#:index-9 #:last-8 #:sum-2 x v)
  (setq v (get-vector user))
  (tagbody (setq #:index-9 -1)
           (setq #:last-8 (length v))
           (setq #:sum-2 0)
       #:L-1 (incf #:index-9)
           (if (not (< #:index-9 #:last-8)) (go oss:END))
           (setq x (aref v #:index-9))
           (setq #:sum-2 (+ #:sum-2 x))
           (go #:L-1)
   oss:END)
  #:sum-2)
```

showS thing &optional (format "~%~S") (stream *standard-output*) => thing

This function is convenient for printing out debugging information while an OSS expression is being evaluated. It can be wrapped around any expression no matter whether it produces an OSS value or a non-OSS value without disturbing the containing expression. The function prints out the value and then returns it. If the value is a non-OSS thing, it will be printed out once at the time it is created. If it is an OSS series thing, it will be printed out an element at a time. The format can be used to print a tag in order to identify the value being shown.

```
(showS format stream)
  == (let ((x thing)) (format stream format x) x)

(letS ((x (Elist '(1 2 3))))
  (Rsum (showS x "Item: ~A, "))) => 6
;;The output "Item: 1, Item: 2, Item: 3, " is printed.
```

permit-non-terminating-oss-expressions

On the theory that non-terminating loops are seldom desired, the OSS macro package checks each loop constructed to see if it can terminate. If this control variable is nil (which is the default), then a warning message is issued for each loop which the OSS macro package thinks has no possibility of terminating. This is useful in the first example below, but not in the second. The form compiler-let can be used to bind this control variable to T around such an expression.

```
(Rlist 4)                   ;Signals warning 15

(block bar                  ;Signals warning 15
  (letS ((x (Eup :by 10)))
    (if (> x 15) (return-from bar x)))) => 20

(compiler-let ((*permit-non-terminating-oss-expressions* T))
  (block bar
    (letS ((x (Eup :by 10)))
    (if (> x 15) (return-from bar x))))) => 20
```

last-oss-loop

This variable contains the loop most recently produced by the OSS macro package. After evaluating (or macro-expanding) an OSS expression, this variable can be inspected in order to see the code which was produced.

last-oss-error

This variable contains the most recently printed warning or error message produced by the OSS macro package. The information in this variable can be useful for tracking down errors.

Side-Effects

The OSS macro package works by converting each OSS expression into a loop. This allows the expressions to be evaluated very efficiently, but radically changes the order in which computations are performed. In addition, off-line ports are supported by code motion. Given all of these changes, it is not surprising that OSS expressions are primarily intended to be used in situations where there are no side-effects. Due to the change in computation order, it can be hard to figure out what the result of a side-effect will be.

Nevertheless, since side-effects (particularly in the form of input and output) are an inevitable part of programming, several steps are taken in order to make the behavior of OSS expressions containing side-effect operations as easy to understand as possible. First, when implicit mapping is applied, it is applied to as large a subexpression as possible. This makes it straightforward to understand the interaction of the side-effects within a single mapped subexpression. Several examples of this are given in the section above which discusses implicit mapping.

Second, wherever possible, the OSS macro package leaves the order of evaluation of the OSS functions in an expression unchanged. Each function is evaluated incrementally an element at a time, but on each cycle, the processing follows the syntactic ordering of the functions in the expression.

The one place where order changes are required is when handling off-line ports. However, things are simplified here by ensuring that the evaluation order implied by the order of the inputs of an off-line function is preserved.

Third, when determining whether or not each termination point is connected to every output in each on-line subexpression, functions whose outputs are not used for anything are considered to be outputs of the subexpression. The reasoning behind this is that if the outputs are not used for anything, then the function must be being used for side-effect and probably matters that the function get evaluated the full number of times it should be. For example, consider the expressions below. The first expression prints out the numbers in a list and returns the first negative number. The second expression signals a warning and the enumeration of the list is duplicated so that the princ will be applied to all of the elements of the list.

```
(letS* ((x (Elist '(1 2 3 -4 5))))
  (princ x)
  (Rfirst-passive (TselectF #'minusp x))) => -4
;;The output "123-45" printed.

(letS* ((x (Elist '(1 2 3 -4 5))))           ;Signals warning 18
```

```
    (princ x)
    (Rfirst (TselectF #'minusp x))) => -4
  ;;The output "123-45" printed.
```

3. Bibliography

[1] A. Aho, J. Hopcraft, and J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading MA, 1974.

[2] G. Burke and D. Moon, Loop Iteration Macro, MIT/LCS/TM-169, July 1980.

[3] R. Polivka and S. Pakin, APL: The Language and Its Usage, Prentice-Hall, Englewood Cliffs NJ, 1975.

[4] G. Steele Jr., Common Lisp: the Language, Digital Press, Maynard MA, 1984.

[5] R. Waters, "A Method for Analyzing Loop Programs", IEEE Trans. on Software Engineering, 5(3):237--247, May 1979.

[6] R. Waters, Synchronizable Series Expressions: Part II: Overview of the Theory and Implementation, MIT/AIM-959, November 1987

[7] Lisp Machine Documentation for Genera 7.0, Symbolics, Cambridge MA, 1986.

4. Warning and Error Messages

In order to facilitate the debugging of OSS expressions, this section discusses the various warning and error messages which can be issued by the OSS macro package while processing the functions described in this document. Error messages describe problems in OSS expressions which make it impossible to process the expression correctly. Warning messages identify less serious situations which are worthy of programmer scrutiny, but which do not prevent the expression from being processed in a way which is, at least probably, correct.

Warning and error messages are both printed out in the following format. Error messages (as opposed to warnings) can be identified by the fact that the word "Error" precedes the message number. (The format is shown as it appears on the Symbolics Lisp machine and may differ in minor ways in other systems.)

```
Warning: {Error} message-number in OSS expression:
containing OSS expression
detailed message
```

For example, the following error message might be printed.

```
Warning: Error 1.1 in OSS expression:
(LETS ((X (ELIST NUMBER-LIST))
     (Y (EUP (CAR HEADER) :TO 4 :LENGTH 5)))
  (RLIST (LIST Y X)))
Too many keywords specified in a call on Eup:
(EUP (CAR HEADER) :TO 4 :LENGTH 5)
```

The first line of each message specifies the number of the warning or error. This number is useful for looking up further information in the documentation below. The next part of the message shows the complete OSS expression which contains the problem. This makes it easier to locate the problem in a program. The remainder of the message describes the particular problem in detail. (The variable *last-oss-error* contains a list of the information which was used to print out the most recent warning or error message.)

The OSS macro package reports problems using warn so that processing of other parts of a program can continue, potentially finding other problems. However, each time an OSS error (as opposed to a warning) is detected, the OSS macro package skips over the rest of the OSS expression without performing any additional checks. Therefore, even if there are several OSS errors in an OSS expression, only one OSS error will be reported. When an OSS error is found, a dummy value is inserted in place of the erroneous OSS expression. As a result, it is virtually impossible for the containing program to run correctly.

The documentation below describes each of the messages which the OSS macro package can produce. Each description begins with a header line containing a schematic rendition of the message. Italics is used to indicate pieces of specific information which are inserted in the message. The number of the warning or error is shown in the left margin at the beginning of the header. For ease of reference, the messages are described in numerical order.

Local errors concerning single OSS functions.
The following error messages report errors which are local in that they stem purely from the improper use of a single OSS function. These errors cover only a few special situations. Many (if not most) local errors are reported directly by the standard Common Lisp processor rather than by the OSS macro package. For example, if an OSS function

is used with the wrong number of arguments, an error message is issued by the standard macro expander.

1.1 Error: Too many keywords specified in call on Eup: call

1.2 Error: Too many keywords specified in call on Edown: call

1.3 Error: Too many keywords specified in call on Tlatch: call

Each of these errors specifies that incompatible keywords have been provided for the indicated function. The entire function call is printed out as shown above.

2 Error: Invalid enumerator arg to TconcatenateF: enumerator

This error is issued if the enumerator argument to TconcatenateF fails to be an enumerator---i.e., fails to be an OSS function that has no OSS inputs, at least one OSS output, and which can terminate.

3 Error: Unsupported &-keyword keyword in defunS arglist.

This error is issued if an &-keyword other than &optional or &key appears in the argument list of defunS. Other keywords have to be supported by using defmacro directly. (See the discussion of defunS.)

4 Error: AlterS applied to an unalterable form: call

This error is issued if alterS is applied to a value which is not alterable. Values are alterable only if they come directly from an enumerator which has an
alterable value, or come indirectly from such an enumerator via one or more transducers which allow alterability to pass through.

5 Error: Malformed lambdaS argument arg.

This error message is issued if an argument of a lambdaS fails to be a valid variable. In particular, it is issued if the argument, is not a symbol, is T or nil, is a symbol in the keyword package, or is an &-keyword. (It is also erroneous for such a variable to be declared special. However, this error is only reported on the Symbolics Lisp Machine.)

6 Error: LambdaS used in inappropriate context: call

This error message is issued if a lambdaS ends up (after macro expansion of the
surrounding code) being used in any context other than as the quoted first argument of a funcallS.

7 Error: Wrong number of args to funcallS: call

This error message is issued if a use of funcallS does not contain a number of arguments which is compatible with the number of arguments expected by the OSS functional argument.

8 Error: Only n return values present where m expected: call

This error message is issued if an OSS function is used in a situation where it
is expected to return more values than it actually does---for example, if a letS tries to bind two values from an OSS function which only returns one, or pass-valS tries to obtain two values from an OSS function which only returns one. (Non-OSS functions return extra values of nil if they are requested to produce more values than they actually do.)

Warnings and errors concerning OSS variables.
The following warnings and errors concern the creation and use of letS and lambdaS variables. Like the errors above, they are quite local in nature and relatively easy to fix.

9 Error: Malformed letS{*} binding pair pair.

This error message is issued if a letS or letS* binding pair fails to be either
a list of a valid variable and a value, or a list of a list of valid variables and a value. The criterion for what makes a variable valid is the same as the one used in Error 5, except that a binding pair can contain nil instead of a variable.

10 Warning: The variable(s) vars declared TYPE OSS in a letS{*}.

This warning message is issued if one or more variables in a letS are explicitly declared to be of type oss. The explicit declarations are ignored.

11 Warning: The letS{*} variable variable is unused in: call

This warning message is issued if a variable in a letS is never referenced in the body of the letS. Note that these variables cannot be referenced inside a nested lambda or lambdaS.

12 Error: The letS{*} variable var setqed.

This error message is issued if a letS variable (either OSS or non-OSS) is assigned to in the body of a letS. It is also issued if any of the variables bound by a lambdaS or defunS are assigned to.

Non-local warnings and errors concerning complete OSS expressions. The following warnings and errors concern non-local problems in OSS expressions. The first two are discussed in further detail in the section on implicit mapping.

13 Warning: Decomposition moves: code out of a binding scope: surround

This warning is issued if the processing preparatory to implicit mapping causes
a subexpression to be moved out of the binding scope for one of the variables in it. The problem can be fixed by using letS to create the binding scope, or by moving the binding form so that it surrounds the entire OSS expression. (The testing for this problem is somewhat approximate in nature. It can miss some erroneous situations and can complain in some situations where there is no
problem. Due to this latter difficulty, the OSS macro package merely issues a warning message rather than issuing an error message.)

14 Error: OSS value carried to non-OSS input by data flow from: call to: call

As illustrated below, this error is issued whenever data flow connects an OSS output to a non-OSS input of an OSS function as in the example below. (If the expression in question is intended to contain a nested loop, the error can be fixed by wrapping the nested portion in a mapS.)

```
Warning: Error 14 in OSS expression:
(Rlist (Rlist (Elist (Elist '((1 2) (3 4))))))
OSS value carried to non-OSS input by data flow from:
(Elist '((1 2) (3 4)))
to:
(Elist (Elist '((1 2) (3 4))))
```

The error message prints out two pieces of code in order to indicate the

source
and destination of the data flow in question. The outermost part of the first piece of code shows the function which creates the value in question. The outermost function in the second piece of code shows the function which receives the value. (Entire subexpressions are printed in order to make it easier to locate the functions in question within the OSS expression as a whole.) If nesting of expressions is used to implement the data flow, then the
first piece of code will be nested in the second one.

15 Warning: Non-terminating OSS expression: expr

This warning message is issued whenever a complete OSS expression appears incapable of terminating. The expression in question is printed. It may well be only a subexpression of the OSS expression being processed. A warning message is issued instead of an error message, because the expression may in fact be capable of terminating or the expression might not be intended to terminate. (This warning message can be turned off by using the variable *permit-non-terminating-oss-expressions*.)

Warnings concerning the violation of restrictions.
The following warnings are issued when an OSS expression violates one of the isolation restrictions or the requirement that within each on-line subexpression, there must be a data flow path from each termination point to each output. In each case, the violation is automatically fixed by the macro package. However, in order to achieve high efficiency, the user should fix the
violation explicitly rather than relying on the automatic fix.

16 Warning: Non-isolated non-oss data flow from: call to: call

This warning is issued if an OSS expression violates the non-OSS data flow isolation restriction. As shown below, the message prints out two pieces of code which indicate the data flow in question.

```
Warning: 16 in OSS expression:
(LETS* ((NUMS (EVECTOR '#(3 2 8)))
      (TOTAL (REDUCEF 0 #'+ NUMS)))
  (RVECTOR (/ NUMS TOTAL)))
Non-isolated non-OSS data flow from:
(REDUCEF 0 #'+ NUMS)
to:
(/ NUMS TOTAL)
```

The OSS macro package automatically fixes the isolation restriction violation by duplicating subexpressions until the data flow in question becomes isolated.
(In the example above, the vector enumeration gets copied.) However, the macro
package is not guaranteed to minimize the amount of code copied. In addition, it is sometimes possible for a programmer to fix an expression much more efficiently without using any code copying. As a result, it is advisable for programmers to fix these violations explicitly, rather than relying on the automatic fixes provided by the OSS macro package.

17.1 Warning: Non-isolated oss input at the end of
the data flow from: call to: call

17.2 Warning: Non-isolated oss output at the start of
the data flow from: call to: call

One of these warnings is issued if an OSS expression violates the off-line port
isolation restriction. The warning message prints out two pieces of code

which
indicate a data flow which ends (or starts) on the port in question. Code copying is automatically applied in order to fix the violation. It is worthwhile to try and think of a more efficient way to fix the violation. As with Warning 16, even if code copying is the only thing which can be done, it is better for the programmer to do this explicitly.

18 Warning: No data flow path from the termination point: call
to the output: call

This warning is issued if a termination point in an on-line subexpression of an
OSS expression is not connected by data flow to one of the outputs. Code copying is automatically applied in order to fix the violation. (However, the OSS macro package has a tendency to copy a good deal more code than necessary.)
The violation can often be fixed much more efficiently by using non-early-terminating OSS functions instead of early-terminating functions or by using Tcotruncate to indicate relationships between inputs.

Errors concerning implementation limitations.
These errors reflect limitations of the way the OSS macro package is implemented rather than anything fundamental about OSS expressions.

19 Error: LambdaS body too complex to merge into a single unit: forms

In general, the OSS macro package is capable of combining together any kind of permissible OSS expression. In particular, there is never a problem as long as
the expression as a whole does not have any OSS inputs or OSS outputs. However, in the body of a lambdaS, it is possible to write OSS expressions which have both OSS inputs and OSS outputs. If such an expression has a data flow path from an OSS input to an OSS output which contains a non-OSS data flow
arc, then this error message is issued. For example, the error would be issued
in the situation below.

```
(funcallS #'(lambdaS (items)                      ;Signals error 19
              (declare (type oss items))
            (Elist (Rlist items)))
          ...)
```

An error message is issued in the situation above, because the situation is unlikely to occur and there is no way to support the situation without resorting to very peculiar code. In particular, the input items in the example
above would have to be converted into an off-line input.

20 Error: The form function not allowed in OSS expressions.

In general, the OSS macro package has a sufficient understanding of special forms to handle them correctly when they appear in an OSS expression. However,
it does not handle the forms compiler-let, flet, labels, or macrolet. The forms compiler-let and macrolet would not be that hard to handle, however it does not seem worth the effort. The forms flet and labels would be hard to handle, because the OSS macro package does not preserve binding scopes and therefore does not have any obvious place to put them in the code it produces. All four forms can be used by simply wrapping them around entire OSS expressions rather than putting them in the expressions.

21--27 Documentation for these errors appears in [6].

5. Index of Functions

This section is an index and concise summary of the functions, variables, and special forms described in this document. Each entry shows the inputs and outputs of the function, the page where documentation can be found, and a one line description.

The names of OSS functions often start with one of the following prefix letters.

E Enumerator.
T Transducer.
R Reducer.

Occasionally, a name will end with one of the following suffix letters.

S Special form.
F Function that takes functional arguments.

In addition, the argument and result names indicate data type restrictions (e.g., number indicates that an argument must be a number, item indicates that there is no type restriction). Plural names are used iff the value in question
is an OSS series (e.g., numbers indicates an OSS series of numbers; items indicates an OSS series of unrestricted values). The name of a series input or
output begins with "O" iff it is off-line.

alterS destinations items => items
Alters the values in destinations to be items.

defunS name lambda-list {doc} {decl}* &body expr-list
Defines an OSS function, see lambdaS.

Ealist alist &optional (test #'eql) => keys values
Creates two series containing the keys and values in an alist.

Edown &optional (start 0) &key (:by 1) :to :above :length => numbers
Creates a series of numbers by counting down from start by :by.

Efile name => items
Creates a series of the forms in the file named name.

Efringe tree &optional (leaf-test #'atom) => leaves
Creates a series of the leaves of a tree.

Ehash table => keys values
Creates two series containing the keys and values in a hash table.

Elist list &optional (end-test #'endp) => elements
Creates a series of the elements in a list.

EnumerateF init step &optional test => items
Creates a series by applying step to init until test returns non-null.

Enumerate-inclusiveF init step test => items
Creates a series containing one more element than EnumerateF.

Eoss &rest expr-list => items
Creates a series of the results of the expressions.

Eplist plist => indicators values
Creates two series containing the indicators and values in a plist.

Esequence sequence &optional (indices (Eup)) => elements
Creates a series of the elements in a sequence.

Esublists list &optional (end-test #'endp) => sublists
Creates a series of the sublists in a list.

Esymbols &optional (package *package*) => symbols
Creates a series of the symbols in package.

Etree tree &optional (leaf-test #'atom) => nodes
Creates a series of the nodes in a tree.

Eup &optional (start 0) &key (:by 1) :to :below :length => numbers
Creates a series of numbers by counting up from start by :by.

Evector vector &optional (indices (Eup)) => elements
Creates a series of the elements in a vector.

funcallS function &rest expr-list => result
Applies an OSS function to the results of the expressions.

lambdaS var-list {decl}* &body expr-list
Form for specifying literal OSS functions.

last-oss-error
Variable containing a description of the last OSS warning or error.

last-oss-loop
Variable containing the loop the last OSS expression was converted into.

letS var-value-pair-list {decl}* &body expr-list => result
Binds OSS variables in parallel.

letS* var-value-pair-list {decl}* &body expr-list => result
Binds OSS variables sequentially.

mapS &body expr-list => items
Causes expr-list to be mapped over the OSS variables in it.

oss-tutorial-mode &optional (T-or-nil T) => state-of-tutorial-mode
If called with an argument of T, turns tutorial mode on.

pass-valS n expr => &rest multiple-value-result
Used to pass multiple values from a non-OSS function into an OSS expression.

permit-non-terminating-oss-expressions
When non-null, inhibits error messages about non-terminating OSS expressions.

prognS &body expr-list => result
Delineates an OSS expression.

Ralist keys values => alist
Combines a series of keys and a series of values together into an alist.

Rand bools => bool
Computes the and of the elements of bools, terminating early.

Rand-late bools => bool
Computes the and of the elements of bools.

Rappend lists => list
Appends the elements of lists together into a single list.

Rbag items => list
Combines the elements of items together into an unordered list.

ReduceF init function items => result
Computes a cumulative value by applying function to the elements of items.

Rfile name items &rest option-plist => T
Prints the elements of items into a file.

Rfirst items &optional (default nil) => item
Returns the first element of items, terminating early.

Rfirst-late items &optional (default nil) => item
Returns the first element of items.

Rhash keys values &rest option-plist => table
Combines a series of keys and a series of values together into a hash table.

Rlast items &optional (default nil) => item
Returns the last element of items.

Rlength items => number
Returns the number of elements in items.

Rlist items => list
Combines the elements of items together into a list.

Rmax numbers => number
Returns the maximum element of numbers.

Rmin numbers => number
Returns the minimum element of numbers.

Rnconc lists => list
Destructively appends the elements of lists together into a single list.

Rnth n items &optional (default nil) => item
Returns the nth element of items, terminating early.

Rnth-late n items &optional (default nil) => item
Returns the nth element of items.

Ror bools => bool
Computes the or of the elements of bools, terminating early.

Ror-late bools => bool
Computes the or of the elements of bools.

Rplist indicators values => plist
Combines a series of indicators and a series of values together into a plist.}

Rsum numbers => number
Computes the sum of the elements in numbers.

Rvector items &key (:size 32) &rest option-plist => vector
Combines the elements of items together into a vector.

showS thing &optional (format "~%~S") (stream *standard-output*) => thing
Displays thing for debugging purposes.

Tchunk amount Oitems => lists
Creates a series of lists of length amount of non-overlapping subseries of Oitems.

Tconcatenate Oitems1 Oitems2 &rest more-Oitems => items
Concatenates two or more series end to end.

TconcatenateF Enumerator Oitems => items
Concatenates the results of applying Enumerator to the elements of Oitems.

Tcotruncate items &rest more-items => initial-items &rest more-initial-items
Truncates all the inputs to the length of the shortest input.

Texpand bools Oitems &optional (default nil) => items
Spreads the elements of items out into the indicated positions.

Tlastp Oitems => bools items
Determines which element of the input is the last.

Tlatch items &key :after :before :pre :post => masked-items
Modifies a series before or after a latch point.

TmapF function &rest items-list => items
Maps function over the input series.

Tmask Omonotonic-indices => bools
Creates a series continuing T in the indicated positions.

Tmerge Oitems1 Oitems2 comparator => items
Merges two series into one.

Tpositions Obools => indices
Returns a series of the positions of non-null elements in Obools.

Tprevious items &optional (default nil) (amount 1) => shifted-items
Shifts items to the right by amount inserting default.

Tremove-duplicates Oitems &optional (comparator #'eql) => items
Removes the duplicate elements from a series.

TscanF {init} function items => results
Computes cumulative values by applying function to the elements of items.

Tselect bools &optional items => Oitems
Selects the elements of items corresponding to non-null elements of bools.

TselectF pred Oitems => items
Selects the elements of Oitems for which pred is non-null.

Tsplit items bools &rest more-bools => Oitems1 Oitems2 &rest more-Oitems
Divides a series into multiple outputs based on bools.

TsplitF items pred &rest more-pred => Oitems1 Oitems2 &rest more-Oitems
Divides a series into multiple outputs based on pred.

Tsubseries Oitems start &optional below => items
Returns the elements of Oitems from start up to, but not including, below.

Tuntil bools items => initial-items
Returns items up to, but not including, the first non-null element of bools.

TuntilF pred items => initial-items
Returns items up to, but not including, the first element which satisfies pred.

Twindow amount Oitems => lists
Creates a series of lists of length amount of successive overlapping subseries.

type oss &rest variable-list
Declaration used to specify that variables are OSS variables.

valS &rest expr-list => &rest multiple-value-result
Returns multiple series values.

PACMAN

By: Michel Denber (Denber.WBST@Xerox.COM)

INTRODUCTION

PACMAN is a Xerox Lisp (Medley, Lyric or Koto) implementation of the arcade game PACMAN. It is fairly faithful to the arcade version, in terms of screen appearance and game dynamics. It runs equally well on Suns, 1132's, 1108's, 1186's, and 1100's (Maikos, Dorados, Dandelions, Doves, and Dolphins). It can run in color for 1132's and 1100's with color boards. Several different methods of user input are supported.

This document describes operational details only. I assume that you know what Pacman is and how to play it.

Load PACMAN.LCOM from your local Lispusers directory. To start the game, type (PACMAN). It will prompt you via menus for input mode, speed, color, and high scores. The menus are described in the next section.

OPTIONS

Input mode: there are currently four ways to control the Pacman, depending on what hardware you have. Although you are asked to pick a mode whenever you start the game, you may switch modes freely at any point during the game.

Mouse - to use this mode, move the mouse in the direction you want the Pacman to go. You only need to move the mouse when you want to change directions.

Keyboard - uses the I, J, K, and space bar as cursor control keys. I is up, J is left, L is right, and space-bar is down. If you look at the arrangement of these keys on the keyboard, you will see that they form an approximate cross. Using the space bar for down (rather than M or ",") lets you control each key comfortably with your right hand; use your thumb for space, and index, middle, and ring fingers for J, I, and L respectively. Note that *you must use the upper-case letters*.

Joystick - this mode lets you connect an Atari joystick to the keyset port on the Alto keyboard of your 1100 or 1132. See the appendix below for details on constructing the interface cable needed. This is the input mode of choice, since the arcade game also uses a joystick. You can, if you want, plug a keyset (if you can find one) into the keyset port, but it is not clear that this offers any particular advantages for data input. I have heard that it is possible to kludge a keyset port on an 1108, but I don't know how to do that.

Voice - perhaps the most exotic input mode, this requires that you have an Interstate Electronics voice recognition board connected to the RS-232 port of your machine. If you possess such a device, call me for specific interfacing details. The commands are "up", "down", "left", and "right". People walking by in the hall will wonder what's going on in your office.

Speed: You can choose one of four speeds for the game. Some effort has gone into making these values similar across different machines, but there's no guarantee they will stay that way. Note that the speed of the game itself will increase (just like the arcade version) as you progress from board to board.

Fast - challenging.

Medium - not unreasonable.

Slow - mellow mode.

Snail - mainly for debugging or the very patient.

Color (y/n): will run the game on your color display (if you have one). The b/w display is blanked out during color mode.

Read high scores (y/n): this option is only available to users on the Xerox Internet, since the high score repository is on this net. If you are an outside or stand-alone user, always answer no to this. The high score list contains the names and score of the top ten players. Note that with the more recent addition of a speed option, this list is somewhat bogus anyway.

Once you have selected your menu options, the game window will appear, along with the message "Insert quarter to start game". You will note a "quarter icon" in the window. Click this to start the game, or click anywhere else to exit. You can skip the introduction screen by clicking the mouse anywhere inside the window while the introduction is in progress.

All of the standard Pacman boards are supported, along with bonus fruit and relative ghost "blue times". You get the correct number of points for eating various objects. On the b/w screen, the ghosts simply invert-video when you eat an energizer; in color they turn blue.

This game is adapted from the author's PDP-11 Fortran version. Call me if you want that one. Please direct all comments, questions, and bug reports to Denber.WBST@Xerox.COM.

KNOWN BUGS

- Pieces of characters are occasionally left lying about the board following collisions.

- Keyboard steering MUST be in uppercase (at least in Medley), lowercase has no effect.

LIMITATIONS

- No sound (that's planned)

- The ghosts do not gain on you as they're chasing you. (You however, can gain on them when you're chasing them).

- The ghosts do not go around turns slower than you do.
- You can't scare ghosts off by feinting at them.
- Only the first intermission is implemented.
- The top ten score probably no longer works, since it depended on an account on {ICE}.
- Color Pacman does not currently work in the Medley release.
- Mouse steering does not currently work in the Medley release.

APPENDIX: PACMAN CONTROL VIA JOYSTICK

1. You must use a switch-closing type joystick (not a pot type). The Atari 2600 (ie. "Video Computer System") or 400/800 computer joystick works fine. The pinouts given below are for this joystick. Signal descriptions are given too, so you can use other similar devices.

2. You will need a male mouse connector (the pins are embedded in the shell, so it's hard to the the sex of this type of connector), a 9 pin male D connector, and a short length of six conductor cable.

3. I will use the following convention for pin numbers since the connectors are not clearly labeled. The views below are from the pin-side of the connectors, ie. not the side that the wires are soldered to.

4. I recommend you check the completed assembly for shorts with a VOM and by running KeyTest (available to 1100's and 1132's from the NetExec) before trying it out on the game. Shorts or opens won't hurt the machine, but they will keep the joystick from working properly (obviously).

Keyset connector:

```
-----------------------------------
 \    1   2   3    4   5   6     /
  \ 7   8   9   10  11  12  13  /
   \ 14  15  16  17  18  19    /
    ---------------------------
```

Atari connector:

```
----------------------
\ 1   2   3   4   5 /
 \  6   7   8   9  /
  -----------------
```

Description:

```
   Atari                    Keyset
Pin   Signal              Pin   Signal

 2    East                9       PAD1
 3    West                10      PAD2
 4    South               11      PAD3
 5    North               12      PAD4
 7    Ground              13      PAD5
 9    Fire                14     Ground
```

Connections:

```
Atari       Keyset
  2           10
  3           11
  4           12
  5           13
  7           14
  9           9    (not used in PACMAN)
```

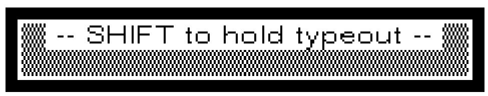
PAGEHOLD

By: Jon L White
Currently maintained by: Bill van Melle (vanMelle.pa@Xerox.com)

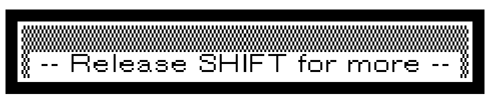
INTRODUCTION

Loading PAGEHOLD.LCOM redefines the function PAGEFULLFN to alter the behavior that occurs when a tty window fills. Rather than inverting the window and waiting indefinitely for type-in, the PAGEHOLD module indicates the hold by an independent notification, and waits for only a specified interval before continuing. Thus, filling the window is no longer a cause for a program to hang indefinitely.

The default behavior of the PAGEHOLD module is to raise a "button" at the corner of the tty window flashing a message, alternating between



and



indicating that output to the window is being held. While in this state, you can release the hold by any of the following means:

Typing any character (of course, the window must own the tty process). This is the same as the old behavior;

Depressing the CTRL key;

Depressing and releasing either SHIFT key;

Clicking with LEFT on the button that announces the hold (clicking instead with MIDDLE gets a menu of options);

Waiting until the timeout has passed (initially, 20 seconds).

When you depress one of the SHIFT keys, the button stops flashing. Output will continue to be held indefinitely as long as one of the SHIFT keys is depressed, even if the timeout passes. If while holding down SHIFT, you depress the CTRL key for a second or so, the button will start flashing again; you may now release CTRL and then SHIFT, and the hold will be maintained without your needing to hold down SHIFT. You can release the hold by any of the means listed above.

If the CTRL key is down when a window fills, output is not held at all. Depressing the CTRL key immediately releases any hold in progress.

The remainder of this document describes ways of tailoring the behavior further.

Controlling the timeout

One of the primary motivations for the PAGEHOLD module is so that printout to a TTY window does not hang indefinitely when one "page" has filled up. The default release time is in the global variable `PAGE.WAIT.SECONDS`, which comes initialized to 20 seconds; a value of 0 causes immediate release (unless a SHIFT key is already depressed). If a window being held has a `PAGE.WAIT.SECONDS` property, then that value is used instead of the global default.

However, if `PAGE.WAIT.SECONDS` is set to `STOP`, then the hold will not be released by any automatic timeout, nor will it be sensitive to the SHIFT or CTRL key actions This mode most closely approximates the current Lisp design, except that a pop-up button signals the hold rather than a video inversion (mousing the button will, nevertheless, still effect a release). The message

"Scrolling Stopped"

appears in the button rather than one of the several "holding" messages.

The Pop-up "Buttons"

A secondary motivation for this facility is to have a pop-up "button" that interactively signals the user of a holding condition on a particular window without obscuring the window's contents, as video inversion does. In addition, the button permits the selective release of a particular window by mousing the button (note that holding down SHIFT, on the other hand, would affect *all* windows currently being held). There are three styles of buttons—`WINKING`, `FLASHING`, and `NIL`—and the selection is determined by the value of the global variable `PAGE.WAIT.ACTIVITY`, which comes initialized to `WINKING`. If a window has a `PAGE.WAIT.ACTIVITY` property, then that value is used instead of the global default, thus allowing different types of buttons on different windows.

A `WINKING` button is a fairly hefty pad—approximately 1/2" by 2 1/2"—which pops up just over the right side of the window's title bar; it will alternately print and clear two short holding messages: one in the upper half of the "button" and one in the lower half. A `FLASHING` button is about the same width, but half the height, and will alternately print the two holding messages. A `NIL` button merely shows the message "Release SHIFT for more".

LEFT-mousing any button causes an immediate release of the hold; MIDDLE-mousing the button brings up a menu offering several options. One of these is "Release this hold!", same as using LEFT. Other menu options permit conversion of the hold to indefinite "hold" or to `STOP` mode; additionally, five options are offered for setting the window's specific `PAGE.WAIT.SECONDS` property.

The `WINKING` button has a different pattern of activity when the hold is placed into indefinite hold mode, but the other button styles do not visibly distinguish this state. If there isn't room to place the button down over the right side of the title bar (because, for example, the window is too close to the screen top), then it will be placed over another corner of the window.

Keyboard Input and Typeahead

Consistent with Lisp's current action, there will be no holding on a window which is its process's TtyDisplayStream *and for which there is typeahead in that process's TTY input buffer*. This action can be overridden by setting `PAGE.WAIT.IGNORETYPEAHEAD` to a non-`NIL` value: typeahead does not inhibit the hold, character input does not release the hold, and no input is ever discarded (note that depressing the SHIFT and/or the CTRL keys does not generate character input). This feature is intended for those who dislike not knowing whether a keystroke will be consumed by the `PAGEFULLFN`—under the default behavior, if the TTY input buffer is empty, then the first character you type will either (a) release a hold already in progress and be discarded, or (b) prevent subsequent holds and be retained, all depending on when exactly you type the character.

PHONE-DIRECTORY

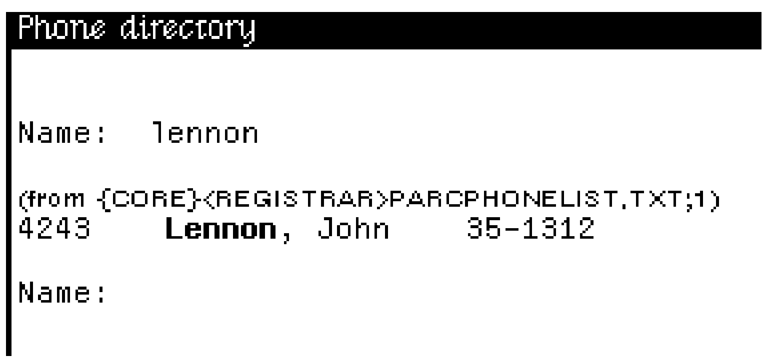
By: smL (Lanning.pa@Xerox.com)

Requires: Lispuser package GREP

INTRODUCTION

The PHONE-DIRECTORY package provides quick and easy access to on–line phone directories.

When you load the PHONE-DIRECTORY package, an icon () will appear on your screen. Opening this icon will start a directory search program (see example below). To look up a person, type in a name, followed by a return. The PHONE-DIRECTORY program will search the available phone lists and print out those entries that contain the name. This can be repeated as many times as you like. When you are done, just SHRINK the window back to its icon.



Example PHONE-DIRECTORY window

Variables

PHONELISTFILES [Variable]

PHONELISTFILES is a list of files that contain phone lists. This is usually set in the INIT.LISP file. The PHONELISTFILES should be unformatted files with a single entry per line. Blank lines are permitted. A typical value for PARC users (as defined in PARC-INIT) is
`({INDIGO}<REGISTRAR>PARCPHONELIST.TXT`
`{INDIGO}<REGISTRAR>ISDNORTHPHONELIST.TXT).`

Phone-Directory-Pos [Variable]

Phone-Directory-Pos is the initial POSITION for the PHONE-DIRECTORY icon. This is defined as an INITVAR in the file, so you can set it before loading the file. The default value is

```
(create POSITION XCOORD ← 15
                 YCOORD ← (DIFFERENCE SCREENHEIGHT 75)).
```

This places the icon in the upper left corner of the screen, just above the exec window.

Phone-Directory-Region [Variable]

Phone-Directory-Region is the initial REGION for the PHONE-DIRECTORY window. This is defined as an INITVAR in the file, so you can set it before loading the file. The default value is

```
(CREATEREGION 15 (DIFFERENCE SCREENHEIGHT 258)
              400 250).
```

This places the window in the upper left corner of the screen.

Notes

When you first open the PHONE-DIRECTORY window, the program will copy the PHONELISTFILES to {CORE}, significantly speeding up queries. Bugging in the title of the PHONE-DIRECTORY window with the left or middle mouse button will produce a menu with an option to recache the files from PHONELISTFILES.

PICK

By: Larry Masinter (lmm@iacm.org)

This document last edited on 4 Aug 2022

INTRODUCTION

When I get frustrated trying to decide what to work on, I use ’pick’ to choose a project. It has a bunch of different choices.

MODULE EXPLANATIONS

(PICK *option choices)* [Function]

pick *option choice1 choice2 ...* [Command]

Option	Meanning	Default choices	Examples
oneof	Choose one of the options	none (must supply)	pick oneof a b c => c
issue	Choose an isue & display it	open issues in Medley	pick issue 3

For now -- the code is commented, so read it :)

Piece-Menus

By: D. Austin Henderson, Jr. (AHenderson.pa@Xerox.COM)

INTRODUCTION

This module provides two solutions to the problem of menus with too many items. Which is useful will depend on the inherent structure of the items. 1) CHUNK-MENUs: For use in which the items have the simple structure of one long list: Break the items up into chunks following the orderingof the items in the list, and then provide a menu which presents one of those pieces and a way of getting other menus containing the other pieces. 2) KEYWORD-MENUs: For use when it is possible to associate keywords with the items: Break the items up into the sets which share the same keyword, and then provide a menu which presents one of those chunks and a way of getting other menus with other keywords. As with standard Interlisp-D menus, these specialized menus are created by a "create" function (cf. (CREATE MENU)), and are used with a single "invoke" function (cf. (MENU menu)). The items and other information are as with standard menus, except that the other information is presented to the create function in the form of a Plist.

CHUNK MENUS

A Chunk Menu appears as a single menu, but is really a data structure encompassing a number of pieces, each represented by a menu, between which the operator can move to find the item desired. Each piece is in three parts: a set of "required items" which will appear in all chunks, a set of indicators for all the chunks, and the items in this piece. Each chunk has up to30 items in it (it can be varied using the CHUNK.COUNT property). If there are fewer than one chunk's worth of items, the CHUNK-MENU behaves just like a standard menu.

(CHUNK.MENU.CREATE *ITEMS PROPERTIES REQUIRED.ITEMS*) [Function]

Creates and returns a Chunk Menu, a data structure of menus each containing some of the (presumably large number of) items.. All items are as with standard menus. The properties understood are: CHUNK.COUNT (see above), TITLE, CENTERFLG, MENUFONT, ITEMWIDTH, ITEMHEIGHT, MENUBORDERSIZE, and MENULAYOUTSIZE. The actual menus are created only when needed.

(CHUNK.MENU.INVOKE *CHUNK.MENU POSITION*) [Function]

Carries out the interaction with the user to select and item from a Chunk Menu. If *POSITION* is non-NIL, the menu will appear at that position, otherwise it will appear under the mouse. All interactions are as with the standard menus. Returns the value produced when a selection is made. Selecting outside any of the menus appearing in the interaction cancels the interaction and returns NIL.

KEYWORD MENUS

A keyword menu appears as a single menu, but is really a data structure encompassing number of menus which the operator can move between to find the item desired. Each piece is in two parts: the set of keywords for all the pieces, and the items in this piece (having this keyword). Each piece Is a Chunk Menu, so that if a particular keyword has many items, its piece is itself broken into pieces. The items in a keyword menu is computed from a list of objects, the things which presumably are being selected among. A function is provided for computing the keywords of each object, and another for computing an item from an object.

(KEYWORD.MENU.CREATE *OBJECTS KEYWORDFN PROPERTIES ITEMFN*) [Function]

Creates and returns a Keyword Menu, a data structure of pieces associated with the keywords of the *OBJECTS* as determined by *KEYWORDFN*. Each piece of the Keyword Menu is a Chunk Menu and contains the items for the objects with the associated keyword. The item is computed from the object by applying the *ITEMFN* to that object; *ITEMFN* should return an item appropriate for standard menus. The menus are determined by the values on the PList *PROPERTIES*. The properties understood are: CHUNK.COUNT (see Chunk Menus),TITLE, CENTERFLG, MENUFONT, ITEMWIDTH, ITEMHEIGHT, MENUBORDERSIZE, and MENULAYOUTSIZE. The actual menus are created only when needed.

(KEYWORD.MENU.INVOKE *KEYWORD.MENU POSITION*) [Function]

Carries out the interaction with the user to select and item from a Keyword Menu. If *POSITION* is non-NIL, the menu will appear at that position, otherwise it will appear under the mouse. All interactions are as with Chunk Menus, which is just that for standard menus for small numbers of items. Returns the value produced by the items when a selection is made. Selecting outside any of the menus appearing in the interaction cancels the interaction and returns NIL.

PLOT

By: Jan Pedersen (pedersen.PA @ Xerox.com)

Uses: TWODGRAPHICS and PLOTOBJECTS

PLOT is a module designed to assist in the production of analytic graphics. PLOT provides automatic scaling, labeling, incremental modification, generalized selection, and a collection of standard graphics primitives which may be combined to produce interactive plots of great diversity.

PLOT is to some degree object-oriented. The primitive components of a plot are plot objects (e.g. points, lines, etc.). A plot manager maintains a display list of plot objects which are individually responsible for displaying themselves, highlighting themselves, etc. The user constructs a plot incrementally, adding plot objects, while the plot manager handles details such as the appropriate scale for the plot. Each plot object is active, in the sense that it is selectable and may have a menu associated with it. In addition, the plot manager may be directed to modify the appearance of the entire plot through a command menu.

The module is open, in the sense that most default behaviors may be overridden by the user, although it is hoped that the defaults will be sufficient for most applications. A functional interface is provided for programmatic access to all of PLOT's facilities.

The plot manager is abstracted as a datatype of type PLOT, along with a collection of functions which operate on PLOT's. Functions are provided to create PLOT's, manipulate their display lists, and modify default menus. Plot objects are abstracted as instances of datatype PLOTOBJECT. A set of default plot objects are provided, along with a mechanism of defining new plot objects.

Plots exist independently of their representation on the screen. Indeed, it is intended that plots may be displayed on ANY imagestream. However, the most common usage is to display a plot in a window, and a PLOT does have an associated WINDOW which may be opened, closed, etc.

Plots may be hard copied, made into image objects, and dumped to file.

The lispuser's module PLOTEXAMPLES contains a few examples of how PLOT may be used to create high level plotting facilities.

BASIC OPERATION

A plot is abstracted as an instance of datatype PLOT which includes a display list, a property list, and an associated window, among other things. PLOT's may be create via the function CREATEPLOT.

(CREATEPLOT *openflg region title border*) [Function]

Returns a PLOT. If openflg is T then the PLOT's associated window is opened with an empty plot. The other arguments are treated as in CREATEW.

An empty plot is initialized to have a world coordinate system extending from 0.0 to 1.0 on either axis, with no labels or tic marks displayed. As objects are added to the plot, the world coordinate system is grown to accommodate the new objects.

A PLOT has an associated window, which is closed by default. The window is used as the primary display device and may be manipulated with the following functions.

(OPENPLOTWINDOW *plot*) [Function]

Opens the plot's associated window.

Returns the associated window.

(CLOSEPLOTWINDOW *plot*) [Function]

Closes the plot's associated window.

(REDRAWPLOTWINDOW *plot*) [Function]

Redraws, by running down the current display list, the contents of the associated window. Opens the window if it is closed.

(GETPLOTWINDOW *plot*) [Function]

Returns the window associated with plot.

(WHICHPLOT *x y*) [Function]

Returns the PLOT associated with the window (or icon) at position (x . y), or at the current cursor position if x and y are defaulted. x may be a WINDOW, in which case the associated PLOT is returned.

A plot object is abstracted as an instance of datatype PLOTOBJECT. A point plot object is an instance of PLOTOBJECT whose data component describes a point. That is, a point plot object is a subtype of PLOTOBJECT; all plot objects satisfy (type? PLOTOBJECT FOO), but only a point plot object satisfies in addition (PLOTOBJECTSUBTYPE? POINT FOO). A collect of standard plot objects has been implemented, including point, curve, polygon, line, and filled rectangle plot objects. The module is designed so that new objects may defined at any time, but that mechanism is described in a separate document.

PLOTOBJECT's may be added to or deleted from a PLOT. The following functions provide an add facility for the standard objects.

(PLOTPOINT *plot position label symbol menu nodrawflg*) [Function]

Only the plot and position arguments are required. Position is a POSITION in world coordinates. Label is an expression which will be PRIN1 'ed whenever a label is required (typically an atom or a string). Symbol is a BITMAP which will be plotted centered at position. The litatoms CROSS, CIRCLE, STAR are bound to convenient BITMAPS. Symbol defaults to STAR. Menu is either a MENU, a litatom, in which case a MENU of that name must be cached on plot (more about this later), or an item list which may be coerced into a MENU.

If nodrawflg is non-NIL then a point object will be added to the display list of plot, but the associated window will not be updated. If Nodrawflg is NIL, and the plot's associated window is not open, it will be opened.

Returns a POINT PLOTOBJECT.

(PLOTPOINTS *plot positions labels symbol menu nodrawflg*) [Function]

As above except that positions is a list of POSITIONS and labels may also be a list. Reasonable things happen if positions and labels are of unequal length.

Returns a list of POINT PLOTOBJECT's.

(PLOTCURVE *plot positions label style menu nodrawflg*) [Function]

The list of POSITION's defines a piecewise linear curve. Style may be an integer which specifies the line width (in pixels) or a list of (linewidth dashing color), any of which may be NIL; defaults to one. For convenience the atoms DOT, DASH and DOTDASH have been bound to a few dashing patterns.

Returns a CURVE PLOTOBJECT.

(PLOTPOLYGON *plot positions label style menu nodrawflg*) [Function]

As in PLOTCURVE, although a polygon is a closed figure

Returns a POLYGON PLOTOBJECT.

(PLOTTEXT *plot position text label font menu nodrawflg*) [Function]

Text should be a STRING to be printed at position.

Returns a TEXT PLOTOBJECT.

(PLOTFILLEDRECTANGLE *plot left bottom width height label*
texture borderwidth menu nodrawflg) [Function]

Texture must be TEXTURE. SHADE1, ..., SHADE8 are bound to some convenient textures. Defaults to SHADE3.

Returns a FILLEDRECTANGLE PLOTOBJECT.

The following two functions add analytic plot objects to the display list of a PLOT. Analytic objects differ from points, curves, etc. by having infinite extents; their appearance on a plot depends on the current world coordinate scale, but adding an analytic object to a plot will not effect the current scale.

(PLOTLINE *plot slope constant label style menu nodrawflg*) [Function]

Slope and constant define an analytic line, y = slope * x + constant. If slope is NIL, it is taken to be infinite; i.e. the line is vertical.

Returns a LINE PLOTOBJECT.

(PLOTGRAPH *plot graphfn nsamples label style menu nodrawflg*) [Function]

Graphfn should be a function of one variable which defines a graph (or the graph of a function) to be drawn on plot. Nsamples is the number of equispaced points along the x-axis of plot at which graphfn is to be sampled when drawn; defaults to 100.

Returns a GRAPH PLOTOBJECT.

Complex objects may be built up from the preceding primitives by defining a compound plot object, which is simply a collection of other plot objects, including other compound objects.

(PLOTCOMPOUND *plot component1 ... componentn typename label*
menu nodrawflg) [NoSpread Function]

A compound plot object is specified by listing its components. In addition, a compound plot object may have its own menu and label. The typename field is supplied to allow different compound objects to be differentiated. Drawing a compound object amounts to drawing its components recursively. In general, operations on compound objects are applied recursively.

Components 1 through n are plot objects. Typename is required and serves to tag this compound object, and is accessable via the function COMPOUNDSUBTYPE. Label and menu are as in other plot objects.

Returns a COMPOUND PLOTOBJECT.

All plot objects may be created independently of the previous functions. This is useful if it is desired to create a plot object without entering it on a PLOT's display list. The following functions create and return the standard plot objects.

(CREATEPOINT *position label symbol menu*) [Function]

Returns a POINT PLOTOBJECT.

(CREATECURVE *positions label style menu*) [Function]

Returns a CURVE PLOTOBJECT.

(CREATEPOLYGON *positions label style menu*) [Function]

Returns a POLYGON PLOTOBJECT.

(CREATETEXT *position text label font menu*) [Function]

Returns a TEXT PLOTOBJECT.

(CREATEFILLEDRECTANGLE *left bottom width height label texture style menu*) [Function]

Returns a FILLEDRECTANGLE PLOTOBJECT.

(CREATELINE *slope constant label style menu*) [Function]

Returns a LINE PLOTOBJECT.

(CREATGRAPH *graphfn nsamples label style menu*) [Function]

Returns a GRAPH PLOTOBJECT.

(CREATECOMPOUND *compoundtype components label menu*) [Function]

Components must be a list of PLOTOBJECT's.

Returns a COMPOUND PLOTOBJECT.

Each PLOT has a display list which is nothing more than a list of plot objects. The display list may be manipulated directly via the following functions.

(ADDPLOTOBJECT *plotobject plot nodrawflg*) [Function]

Interns plotobject on the display list of plot, and updates the associated window. The update is suppressed if nodrawflg is non NIL.

One might think of PLOTPOINT as being equivalent to:

```
(ADDPLOTOBJECT  (CREATEPOINT position ....)  plot  nodrawflg)
```

Interns plotobject on the display list of plot, and updates the associated window. The update is suppressed if nodrawflg is non NIL.

Returns plotobject.

(DELETEPLOTOBJECT *plotobject plot nodrawflg nosaveflg*) [Function]

Deletes plotobject from the display list of plot, and updates the associated window accordingly. The update is suppressed if nodrawflg is T. If nosaveflg is T, then the deleted objected will not be saved for possible later undeletion.

Returns plotobject if it was deleted from the display list, else NIL.

A PLOT has collection of properties, some of which are maintained by the plot manager, and others which may be used to cache arbitrary user data. All plot properties are accessed via the function PLOTPROP.

(PLOTPROP *plot prop newvalue*) [NoSpread Function]

If newvalue is absent then the current value of prop is returned. If newvalue is supplied (even if it is NIL) then the value of prop is set and the old value returned. The distinguished prop's PLOTOBJECTS, PLOTSCALE, SELECTEDOBJECT, PLOTWINDOW, PLOTWINDOWVIEWPORT, PLOTPROMPTWINDOW, and PLOTSAVELIST refer system maintained properties plot, and should be treated as read only. Compiles open in some cases.

For example, The display list of plot may be accessed by the expression.

```
(PLOTPROP plot 'PLOTOBJECTS)
```

For convenience in manipulating the property list of a PLOT, the following functions are provided.

(PLOTADDPROP *plot prop itemtoadd firstflg*) [Function]

If the value of prop is a list then itemtoadd is added to the end of the list. If the value of prop is NIL, it is set to (LIST itemtoadd). Firstflg indicates that the new item is to be the first in the list rather than the last. Works only for user defined properties.

Returns the new value.

(PLOTDELPROP *plot prop itemtodelete*) [Function]

If itemtodelete is a member (MEMB) of the prop value, it is deleted. Works only for user defined properties.

Returns NIL if nothing was deleted, else the new value of prop.

(PLOTREMPROP *plot prop*) [Function]

Destructively removes prop from property list of plot. Works only for user defined properties.

Each plot object also has a property list. As with PLOT's, some of the properties are maintained by the system, but the rest may be used to store arbitrary data objects. The property list of a plot object is accessed through the function PLOTOBJECTPROP.

(PLOTOBJECTPROP *object prop newvalue*) [NoSpread Function]

As in PLOTPROP. The distinguished props are OBJECTMENU, OBJECTLABEL, and OBJECTDATA. The property, OBJECTMENU, may be set as well as read; if the newvalue is a list of items, it will be coerced into a menu.

(PLOTOBJECTADDPROP *object prop itemtoadd firstflg*) [Function]

As in PLOTADDPROP. Firstflg indicates that the new item is to be the first in the list rather than the last.

(PLOTOBJECTDELPROP *object prop itemtodelete*) [Function]

As in PLOTDELPROP.

DEFAULT MOUSE BUTTON ACTIONS

The user may interact with a plot through its associated window. A plot provides two default menu's, the RIGHT menu, which pops up if the right mouse button is depressed within a plot's window, and typically contains items relevant to the plot as a whole, and the MIDDLE menu, which pops up if the middle mouse button is depressed, and typically contains items relevant to the currently selected plot object. The left mouse button is used exclusively for selection. The right menu may optionally be fixed to the right hand side of the plot window for easy reference. In summary:

Left Button

While depressed will select the closest plot object.

Middle Button

Pops up a menu of default actions on the selected object

Right Button

Pops up a menu of default actions on the plot as a whole

DEFAULT MIDDLE MENU ITEMS

Label

Label the selected object. Either a default location for the label is selected (for point plot objects), or the user is queried for a location.

Unlabel

If the object is label, remove the label.

Relabel

Change the object's label

Delete

Remove the object from the plot. May be undeleted later.

DEFAULT RIGHT MENU ITEMS

Layout

Create a SKETCH of the contents of the PLOT. Requires SKETCH and SKETCHSTREAM to be loaded.

Redraw

Redraw the plot

Rescale

Compute a new scale for both the X and the Y axis based on the objects currently displayed. May also rescale the X or Y axis separately.

Extend

Extend the axes slightly on either side so plot objects occuring on the borders may become visible. May be applied separately to either axis.

Labels

Change the marginal labels. May either Choose a margin explicitly, or respond to query.

Tics

Enable or disable marginal tics.

Undelete

Restore the last plot object deleted. Subsidiary items allow selected objects to be restored.

Deselect

Deselects the current selected object.

The default menus may be altered or superceded altogether. Each plot object may either use the default middle menu, another cached menu, or provide its own individual menu.

Menus are described by item lists of the form (label function helpstring [(subitems ....)]). Function may be a litatom in which case the function is called with one argument, plot, for right menu items,or two arguments, plotobject and plot, for all other menus. If function is a list the CAR of the list is a APPLIED to `(CONS PLOTOBJECT (CONS PLOT (CDR list)))`, etc.

The following functions facilitate modifying existing menus, and creating new menus.

(PLOTMENU *plot menuname newmenu*) [NoSpread Function]

Plot and menuname are required. If newmenu is not present, then the current value of menu menuname is returned. Menuname may be RIGHT or MIDDLE, in which case the default menus are referred to, or any LITATOM, in which case the cached menu by that name is referred to. Menus other than RIGHT or MIDDLE will typically be specialized menus for particular plot objects. If present, newmenu must be a MENU.

(PLOTMENUITEMS *plot menuname menuitems*) [NoSpread Function]

Plot and menuname are required. If menuitems is not present, then the current item list for the MENU menuname is returned. If menuitems is present, then menu menuname is replaced with a new menu with items list menuitems. All the properties (if any) of the old menu are copied over. Menuname may be one of RIGHT or MIDDLE, in which case the operations refer to the default right or middle mouse button menus or any other LITATOM, in which case the operations refer to a menu cached on plot by that name. Menus other than RIGHT or MIDDLE will typically be specialized menus for particular plot objects.

(PLOTADDMENUITEMS *plot menuname itemstoadd*) [Function]

Itemstoadd must be a list of menu items. Adds each item in itemstoadd to the end of the item list for menu menuname and replaces menu menuname with a new MENU having the appropriate item list.

Returns the the new item list for menuname.

(PLOTDELMENUITEM *plot menuname itemstodelete*) [Function]

Itemstodelete must be a list of items. For each element of itemstodelete, if it is a LITATOM, then deletes the item whose CAR is EQ to it. If it is a LISTP, then deletes the item EQUAL to it. Replaces menu menuname with a new MENU having the appropriate item list.

Returns NIL if no items were deleted, else the new item list.

(PLOT.FIXRIGHTMENU *plot fixedflg*) [NoSpread Function]

Fixedflg is optional. If not present that the current state of the right menu of plot is returned; T implies the right menu is fixed. If Fixedflg is supplied the right menu state is correspondingly changed.

The middle button menu for a particular plot object is a property of that plot object, and may be accessed via the function PLOTOBJECTPROP. For example, the expression,

```
(PLOTOBJECTPROP object 'OBJECTMENU)
```

will return the current middle button menu for object. If the OBJECTMENU property is NIL, then the system default MIDDLE menu is used, if it is a LITATOM, than a specialized cached menu by that name is used, finally, if it is a MENU, then that menu is used.

Two default fonts are provided, a large font for labels and a small font for tic marks. Both may be reset and that aspect of a plot will change accordingly with the next redraw.

LARGEPLOTFONT [Variable]

Default value: (Gacha 12 BRR)

SMALLPLOTFONT [Variable]

Default value: (Gacha 8 MRR)

Detailed Operation

Most visible aspects of a PLOT may be changed programmatically. The following functions allow the user to specify labels, etc., as well as override the default algorithms for drawing tics, etc.

(PLOTLABEL *plot margin newlabel nodrawflg*) [NoSpread Function]

Plot and position are required. Margin must be one of TOP, BOTTOM, LEFT, OR RIGHT. If newlabel is absent, then the current margin label is returned (may be NIL). If newlabel is present then the margin label is set to newlabel. The display is automatically updated unless nodrawflg is non NIL.

(PLOTTICS *plot margin newvalue nodrawflg*) [NoSpread Function]

Plot and margin are required. Margin must be one of TOP, BOTTOM, LEFT, OR RIGHT. If newvalue is absent, returns the tic status of that margin. NIL implies no tics or labels, T implies both. If newvalue is present, then sets margin's tic status. The display is automatically updated unless nodrawflg is non NIL.

The appearance of the tic marks will also depend on the tic generation method employed. The default is simply to make down tics at "pretty" intervals from the max to the min of each axis in world coordinates. However, non-numeric tic marks, and other behaviors are user specifiable by the function PLOTTICMETHOD.

(PLOTTICMETHOD *plot margin newmethod nodrawflg*) [NoSpread Function]

Plot and margin are required. Margin must be one of TOP, BOTTOM, LEFT, OR RIGHT.If newmethod is absent, returns the current tic method for margin margin. Newmethod may be one of NIL, implying the default tic method, a list of CONS pairs (value . label), in which case label (if non-NIL) will be printed at value, or a list of numbers, which is equivalent to ((value . value) ...) or a function which will be called with args, margin plotscale plot, and should return a list as above. Plotscale is a datatype which descibes the current scale of the plot.

(DEFAULTTICMETHOD *margin plotscale plot*) [Function]

The result depends on the ticinfo field of plotscale, which should be an instance of the PLOTSCALE datatype. The ticinfo field will be an instance of datatype TICINFO. If its ticinc field is a number (the usual case) then it returns a list of numbers, starting at ticmin and ending at ticmax in increments of ticinc, otherwise returns ticinc (should be a list).

When a plot object is added to a plot, the scale of the plot is adjusted so that the object is visible. This is accomplished by comparing the extent (in world coordinates) of the object with the current scale of the plot. If the scale needs to be enlarged, a new interval is chosen for each axis which is guaranteed to include the object and also be some multiple of a "round" increment -- in other words, a pretty tic interval. The default behavior of this scaling algorithm may be altered in several ways.

The pretty tic interval is determined by the TICFN for each axis. The default uses the function SCALE to find a suitable interval. This may be altered by supplying a TICFN other than the default.

Given a pretty tic interval, the default is to simply use the end points of that interval as the endpoints of the scale for each axis. This may be altered by supplying a SCALEFN other than the default.

In other words the actually displayed interval (for each axis) in world coordinates (what I will call the plot interval) is separated from the pretty tic interval (for each axis). The pretty tic interval is computed first, then the plot interval is computed in the presence of that information. This separation is useful if the user wishes to plot objects in a coordinate system different from the one used to display tic marks.

The current state of each axis of a PLOT is cached in the plot property plotscale, whose value is an instance of datatype PLOTSCALE. A PLOTSCALE has three fields for each axis, one which contains an instance of AXISINTERVAL, describing the actual plot interval for that axis, another which contains an instance of TICINFO, which describes the pretty tic interval for that axis, and a third which is a simply a place to cache a user supplied TICFN and SCALEFN.

(PLOTTICFN *plot axis ticfn nodrawflg*) [NoSpread Function]

Ticfn is optional. If not present the current ticfn for the indicated axis is returned. If supplied, the state of that axis is correspondingly updated. A ticfn is called with args min, max, and plot and should return an instance of TICINFO. If the state of plot is changed, the appropriate axis is rescaled. A value of NIL implies the default ticfn.

(DEFAULTTICFN *min max* -- -- --) [Function]

The default ticfn for each axis. Uses the function SCALE to find a suitable pretty tic interval.

(PLOTSCALEFN *plot axis scalefn nodrawflg*) [NoSpread Function]

Scalefn is optional. If not present the current scalefn for that axis of plot is returned. If supplied, the state of that axis is updated. A scalefn is called with four arguments, the min and max extent (in world coordinates) on that axis of the plotobjects currently displayed, the TICINFO for that axis, and the plot;

the scalefn should return an AXISINTERVAL which will determine the scale for that axis of plot. A value of NIL implies the default scalefn.

(DEFAULTSCALEFN *min max ticinfo*) [Function]

The default scalefn for each axis.

Returns an AXISINTERVAL with endpoints identical to the endpoints of ticinfo.

(ADJUSTSCALE? *extent plot*) [Function]

Determines whether extent will fit into the current viewing area of plot. If so, returns NIL. If not, returns T and updates the plotscale of plot.

(EXTENTOFPLOT *plot*) [Function]

Computes the current extent of plot by mapping EXTENTOBJECT down the display list. Returns an EXTENT.

To be precise, the scaling algorithm operates as follows; a min and max extent of the data is computed (via EXTENTOFPLOT or entered manually in the case manual rescaling), then CHOOSETICS is called, which returns an instance of TICINFO. CHOOSETICS either uses a default TICFN, or one supplied by the user, The default TICFN, calls SCALE repeatedly to find an "optimal" tic interval in world coordinates. Once the TICINFO instance has been computed, CHOOSESCALE is called with the original min, max and the TICINFO, and returns an instance of AXISINTERVAL, which will determine the actually displayed plot interval. Again, CHOOSESCALE either uses a default SCALEFN, or one supplied by the user. The default SCALEFN simply uses the end points of the passed in pretty tic interval as the end points of the AXISINTERVAL which it returns. Finally, the PLOT is redrawn with the new scale -- notice that the plot interval may either be larger or smaller than the pretty tic interval; the margin drawing routines are robust enough to deal with all cases.

For example, suppose the world coordinates are in centigrade and it is desired to produce a pretty tic interval in units of Fahrenheit (this is an easy case since the transformation between scales is linear -- more about that later). The user would then supply a TICFN which would transform the incoming min and max to Fahrenheit , apply the default TICFN on the transformed min and max, obtain a TICINFO in Fahrenheit, transform the fields of that record back to Centigrade, and return that record. Note, it is always assumed that the fields of a returned TICINFO are in the units of the world coordinate system. The rest of the machinery would then go through as before.

A tricker example is one in which it is desired to produce unequispaced tic marks. Suppose the data were plotted on a log scale (that is, log was applied BEFORE plotting the data).The default algorithm would produce a pretty tic interval in the log scale. It might be desired instead to produce one pretty in the original scale. The user would then supply a TICFN which would exponentiate the incoming min and max, apply the default TICFN on the transformed min and max, obtain a TICINFO in the original scale, then return a TICINFO in the logscale. Note; since equispaced tic marks in the orginal scale are not equispaced in the log scale, the TICINC field of the returned TICINFO would be a list of the unequispaced tic marks values, rather than a number.

The plot scale of each axis may be manipulated directly through the following functions.

(PLOTAXISINTERVAL *plot axis newinterval nodrawflg*) [Function]

Plot and axis are required. Axis must be one of X, or Y. If newinterval is NIL , returns the current AXISINTERVAL for that axis. If newinterval is non-NIL it must be an AXISINTERVAL.

(PLOTTICINFO *plot axis newticinfo nodrawflg*) [Function]

Plot and axis are required. Axis must be one of X, or Y. If newticinfo is NIL , returns the current TICINFO for that axis. If newticinfo is non-NIL it must be a TICINFO.

On occasion it is useful to clean out an existing plot instead of creating a new one.

(PLOT.RESET *plot xscale yscale flushmargins flushprops nodrawflg*) [Function]

Returns plot to a pristine state. If xscale and yscale are provided, the scale of the plot is set accordingly.

Finer control over the behavior of plot objects is possible through the following functions.

(TRANSLATEPLOTOBJECT *plotobject dx dy plot nodrawflg*) [Function]

Moves plotobject dx, dy in world coordinates and updates the associated window accordingly. The update is suppressed if nodrawflg is non NIL.

(DRAWPLOTOBJECT *plotobject plot*) [Function]

Draw plotobject in the window asssociated with plot. As with all the display functions, the window should be opened beforehand. DRAWOBJECT does NOT check that the window is open.

APPLY's the plotobject's DRAWFN.

(ERASEPLOTOBJECT *plotobject plot*) [Function]

APPLY's the plotobject's ERASEFN

(HIGHLIGHTPLOTOBJECT *plotobject plot*) [Function]

Invoked when a plotobject is selected

(LOWLIGHTPLOTOBJECT *plotobject plot*) [Function]

Invoked when a plotobject is deselected

(EXTENTOFPLOTOBJECT *plotobject plot*) [Function]

Computes the extent of plotobject in world coordinates.

Returns an EXTENT, which has fields MAXX, MINX, etc.

(DISTANCETOPLOTOBJECT *plotobject streamposition plot*) [Function]

Returns the "distance" to plotobject from streamposition in stream coordinates. Value returned may be a FIXP or a FLOATP, but is always a distance in stream coordinates.

(CLOSESTPLOTOBJECT *plot streamposition*) [Function]

Returns the "closest" plotobject on plot's display list to streamposition.

(DESELECTPLOTOBJECT *plot*) [Function]

Deselects the current selected object of plot

Plot objects also have "afterfns". That is, functions which are optionally invoked after some standard operation. These are stored as plot object properties with distinguished names, and invoked with at least two args, the plotobject and the plot.

WHENADDEDFN [Property]

The WHENADDEDFN is called with three arguments, plotobject, plot, and nodrawflg

WHENDELETEDFN [Property]

The WHENDELETEDFN is called with four arguments, plotobject, plot, nodrawflg, and nosaveflg.

WHENDRAWNFN [Property]

The WHENDRAWNFN is called with three arguments, plotobject, viewport and plot.

WHENERASEDFN [Property]

WHENHIGHLIGHTEDFN [Property]

WHENLOWLIGHTEDFN [Property]

WHENTRANSLATEDFN [Property]

A PLOT has two associated windows, the mainwindow in which the graphics, labels, tics, etc. are displayed and an attached promptwindow. The mainwindow is cached as plot property and may be accessed via the function PLOTPROP. A function is provided for easy access to the prompt window.

(PLOTPROMPT *text plot*) [Function]

Text is output in the one character high prompt window of plot.

PLOT's may be drawn in ANY imagestream (but only interacted with in the PLOT's associated window). The following function is the fundamental draw primitive.

(DRAWPLOT *plot stream streamviewport streamregion*) [Function]

Stream is any imagestream. Streamviewport is a viewport on that stream that defines the the world to stream transformation. Streamregion is a region in stream coordinates that will contain the entire image (for a window it will be the CLIPPINGREGION). Streamviewport is usually the result of ADJUSTVIEWPORT.

For more information about viewport, consult the documentation for the **TWODGRAPHICS** module.

(ADJUSTVIEWPORT *viewport streamregion plot*) [Function]

Viewport is a VIEWPORT whose parentstream is the imagestream of interest. Streamregion is a region in stream coordinates that will contain the entire image.

Adjusts the Streamsubregion and Worldregion of viewport to reflect the current scale and margin setting of plot.

(MINSTREAMREGIONSIZE *stream plot*) [Function]

Returns a CONS pair (minwidth . minheight) of the plot in stream coordinates.

A plot has "afterfns" for two major operations, opening and closing the plotwindow. These are stored as plot properties with distinguished names. The values of these properties may be a single function or a list of functions which are called in sequence with the plot as an argument.

WHENOPENEDFN [Property]

WHENCLOSEDFN [Property]

PLOT's may be copied, made into image objects, dumped onto files, sent in the mail, etc.

(COPYPLOT *plot*) [Function]

Returns a copy of plot. The user defined properties require special handling. If there exists a plot prop COPYFN, which may be function or list of functions, the function (or functions) will be invoked with the arguments newplot plot and propname for each user defined property on plot. If the function returns a non-NIL value, it will be used as the value of propname on newplot. In the case of a list of functions, the first non-NIL value (traveling from the head to the tail of the list of functions) will be used as the new prop value. Otherwise the prop will be HCOPYALL'ed.

(COPYPLOTOBJECT *plotobject plot*) [Function]

Returns a copy of plotobject. The protocol for copying objectprops is similar to plot props. The plotobject may have a COPYFN prop which may be a function or list of functions. The function (or functions) will be invoked with the arguments newplotobject plotobject plot propname. The first non-NIL value will be used as the prop value else the property will be HCOPYALL'ed.

(PRINTPLOT *plot stream*) [Function]

Writes out an HREADable symbolic representation of plot on stream. Again, user defined properties require special handling. If there exists a plot prop PUTFN, which may be function or list of functions, the function (or functions) will be invoked with the arguments plot propname and stream for each user defined property on plot. If the function returns a non-NIL value, it is assumed an HREADable representation of the prop value has been written out on stream. In the case of a list of functions, the functions will invoked one at time, starting from the head of the list, until a non-NIL result is obtained. If there is no PUTFN, or the function (or none of the functions) returns a non-NIL value, the prop is HPRINT'ed.

Lists of the form ((FUNCTION function) arg) are recognized by the inverse of PRINTPLOT, READPLOT, to imply that function should be called with plot and arg as arguments at HREAD time, and the value returned to be the prop value.

(PRINTPLOTOBJECT *plotobject plot stream*) [Function]

Writes out an HREADable symbolic representation of plotobject on stream. As in PRINTPLOT user defined object properties require special handling. The protocol is the same as in PRINTPLOT.

The following data types have HPRINT macros and need no special handling: FONTDESCRIPTOR, MENU, PLOT, and PLOTOBJECT.

A file package command has been defined to simplyfy dumping PLOT's on files.

(PLOTS . *plots*) [FilePkgCom]

The syntax is identical to VARS.

A plot image object is fully supported.

(CREATEPLOTIMAGEOBJ *plot*) [Function]

Returns an image object which contains a copy of plot. These image objects can also be created by copy-selecting from a plot window into a host window (e.g. TEdit or Sketch) that supports image objects. Such a selection will ask whether the plot should be inserted as a bitmap or a plot, the latter case constructing a plot image object. Buttoning on the image object provides the option of reshaping the plot or creating a separate plot window in which the plot can be modified. Closing the plot window will ask whether the new plot should be reinserted in the host.

PLOTBOXPATCH

By: Tad Hogg (Hogg.PA@Xerox.COM)

Uses: PLOT

This document last edited on 26-Jan-89.

INTRODUCTION

This file redefines the plot functions BOXREGION and DRAWPLOT so that each side of the box around the plot need not be drawn. Specifically, giving a plot a non-NIL value for the properties NOLEFT, NORIGHT, NOTOP or NOBOTTOM will eliminate the respective side from the drawing of the plot.

PLOT EXAMPLES

By: Jan Pedersen (Pedersen.PA @ Xerox.com)

Uses: PLOT

This module contains two examples of how PLOT might be used to produce high level plotting facilities. The first example is a histogram primitive, and the second is a scatterplotter. The code is commented, and exploits most of the facilities in PLOT. The scatterplot example is the simpler of the two, and is suggested as a starting point.

(SCATPLOT *y x pointlabels ylabel xlabel title symbol*) [Function]

Generates of a scatterplot of y vs x which are numeric lists of equal length. If x in NIL, then y is plotted vs the integers from 1 to (LENGTH y). Pointlabels is a list of labels, one for each point plotted. Ylabel and xlabel are labels for the x and y axis respectively. Title is a title for the scatterplot. Symbol is the plotting symbol to use, must be a BITMAP; defaults to STAR.

Returns a PLOT.

(HISTPLOT *batch label shade*) [Function]

Batch is a list of numbers, or a list of pairs (number . frequency) whose histogram will be displayed. Label is an optional label for those numbers. Shade is a shade to use to fill the bars of the histogram (defaults to SHADE3). The case of all entries in batch being integers is treated specially.

Returns a PLOT.

PLOT OBJECTS

By: Jan Pedersen (pedersen.PA @ Xerox.com)

Uses: PLOT and TWODGRAPHICS

Plot objects are the primitive quantities of the **PLOT** module. A plot object is abstracted as an instance of datatype PLOTOBJECT. A point plot object is an instance of PLOTOBJECT whose data component describes a point. That is, a point plot object is a subtype of PLOTOBJECT; all plot objects satisfy `(`**`type?`**` PLOTOBJECT FOO)`, but only a point plot object satisfies in addition `(PLOTOBJECTSUBTYPE? POINT FOO)`.

A PLOTOBJECT is both a datatype and a collection of functions that implements a set of generic operations on that plot object. A plot object must know how to draw itself, erase itself, highlight itself, etc. The PLOT module then deals only with generic operations, and allows the plot objects to implement them as is appropriate.

PLOTOBJECT [Datatype]

OBJECTFNS [Field]

Must be an instance of PLOTFNS

OBJECTSUBTYPE [Field]

Describes the plot objects subtype

OBJECTUSERDATA [Field]

Space for a propery list

OBJECTMENU [Field]

The object's MENU

OBJECTLABEL [Field]

Something to print

OBJECTDATA [Field]

Space for a datatype that describes the subtype of this PLOTOBJECT

The field OBJECTFNS must be an instance of PLOTFNS, essentially a vector of functions which implements the generic operations.

PLOTFNS [Datatype]

DRAWFN [Field]

Implements the DRAWOBJECT generic operation

ERASEFN [Field]

etc.

HIGHLIGHTFN [Field]

LOWLIGHTFN [Field]

LABELFN [Field]

MOVEFN [Field]

EXTENTFN [Field]

DISTANCEFN [Field]

COPYFN [Field]

PUTFN [Field]

GETFN [Field]

The generic operations are:

(DRAWPLOTOBJECT *object viewport plot*) [Function]

Draw the object within viewport. A VIEWPORT may be thought of as a sub imagestream. It will usually be associated with the plot's PLOTWINDOW, but might might also be associated with some other image stream. Typically this generic operation will make use of functions from TWODGRAPHICS and the position of the object in world coordinates. The plot is also passed as an argument, so that the draw operation may make use of information cached on the property list of plot.

The only operation that is expected to draw on streams other than the PLOTWINDOW is drawobject, so the drawfn may have to behave differently depending on the imagestreamtype of the stream. All other generic operations are assumed to operate on the PLOTWINDOW. The idea here is that plot's may be drawn on any stream, but may be interacted with only through the PLOTWINDOW. It is also guaranteed that an object will be drawn before it is erased, highlighted, etc.

(ERASEPLOTOBJECT *object viewport plot*) [Function]

Erase the object from the viewport. The inverse of DRAWOBJECT. It is guaranteed that the viewport will be on the PLOTWINDOW

(HIGHLIGHTPLOTOBJECT *object plot*) [Function]

Highlight the object. Used in selection.

(LOWLIGHTPLOTOBJET *object plot*) [Function]

The inverse of HIGHLIGHTOBJECT. With XOR drawing the HIGHLIGHTFN and the LOWLIGHTFN can often be the same.

(MOVEPLOTOBJECT *object dx dy plot*) [Function]

Destructively alter the object's OBJECTDATA, so that its position is moved dx, dy units (in world coordinates).

(LABELPLOTOBJECT *object plot*) [Function]

If it is desired to label the object, the LABELFN will be called. Often the function LABELGENERIC will do the trick.

(EXTENTOFPLOTOBJECT *object plot*) [Function]

Should return an EXTENT, which expresses the range of the object in world coordinates.

EXTENT [Datatype]

MINX [Field]

Minimun extent in the X (horizontal) direction

MAXX [Field]

Maximun extent in the X (horizontal) direction

MINY [Field]

Minimun extent in the Y (vertical) direction

MAXY [Field]

Maximun extent in the Y (vertical) direction

All fields are type floating.

(DISTANCETOPLOTOBJECT *object streamposition plot*) [Function]

Should return a number (more efficient if it returns a SMALLP), which is some measure of the distance from the REPRESENTATION of the object to the POSITION streamposition. Note that distance is calculated in stream coordinates, NOT world coordinates. This is done for efficiency and logical consistency. Selection makes most sense as an activity in stream coordinates.

A plot object will typically cache its stream coordinates when it is drawn. Although not strictly necessary (it is always possible to backsolve to stream coordinates from world coordinates), this improves efficiency many fold by avoiding generation of floating point boxes.

The following functions are provided to allow the plot object to customize how it is copied, printed on file, etc. The generic defaults will usually be satisfactory.

(COPYPLOTOBJECT *object plot*) [Function]

Returns a copy of object. COPYOBJECT will create a new instance of PLOTOBJECT and copyover all the fields of object except for OBJECTDATA. The object's COPYFN is evoked with the agruments object and plot and is expected to return a new instance of OBJECTDATUM. The objects property list is handled as follows: If object has a prop COPYFN (which may be a function or list of functions), for each property it is called with the arguments newobject, oldobject, plot, propname. If the returned value is non-nil it is used as the value for that property on newobject; else the prop value is HCOPYALL'ed. If the value of COPYFN is a list of functions, they are invoked in order head to tail, and the first non-NIL value is used as the new value.

(PRINTPLOTOBJECT *object plot stream*) [Function]

Writes out to stream an HREADable symbolic representation of object. As in COPYOBJECT, PRINTOBJECT takes care of all PLOTOBJECT fields except of OBJECTDATUM. The objects PUTFN will be invoked with the arguments object plot stream and is expected to write out a representation of OBJECTDATUM which is HREADable. This will usually be in prop list format.

Again the prop list of object requires special handling. The special object prop PUTFN may be a function or list of functions. For each property it will be invoked with the arguments object plot propname and stream and if it returns a non-NIL value, it is assumed that property has been written out in a HREADable format. Again, if the the PUTFN prop is a list of fns then if any one of them returns non-NIL then the property is assumed written out. If there is no PUTFN then the property is (HPRINT prop stream NIL T) 'ed.

PUTFNS may put out special lists of the form ((FUNCTION fnname) arg) in which case fnname will be invoked at HREAD time with args object plot propname arg and fnname will be expected to return the propvalue of propname.

(READPLOTOBJECT *stream*) [Function]

Reads in the product of PRINTOBJECT. Calls the objects GETFN to read in the OBJECTDATA field.

An instance of PLOTFNS may be created by the function:

(CREATEPLOTFNS *drawfn erasefn extentfn distancefn highlightfn lowlightfn labelfn movefn copyfn putfn getfn borrowfrom*) [Function]

Returns an instance of PLOTFNS. Drawfn, erasefn, and extentfn are required. If a distancefn is supplied then so must be a highlightfn. Lowlightfn defaults to highlightfn, labelfn defaults to LABELGENERIC. The other arguments also default to some safe, if not too efficient genericfn.

A primitive inheritance scheme is implemented via the optional argument borrowfrom. If supplied, borrowfrom must be an instance of PLOTFNS. Before creating the new instance of PLOTFNS, the NIL arguments passed are filled in from the fields of borrowfrom, with the following exception; lowlightfn is only inherited if highlightfn is also NIL.

The OBJECTDATA field will typically be a datatype which holds the data characterizing the PLOTOBJECT. For example a point plot object will have an OBJECTDATA field whose value is an instance of the datatype POINTDATA (has fields position, symbol, etc). So, a point PLOTOBJECT is a specialization of PLOTOBJECT. The field OBJECTSUBTYPE is supplied to make the subtype explicit. The following macro is provided to facilitate testing for plot object subtypes.

(PLOTOBJECTSUBTYPE? *subtype plotobject*) [Macro]

Essentially tests if (EQ subtype (fetch OBJECTSUBTYPE of plotobject))

(PLOTOBJECTSUBTYPE *plotobject*) [Function]

Returns the value of the OBJECTSUBTYPE field.

PLOTOBJECTS may be created via the function:

(CREATEPLOTOBJECT *objectfns objectlabel objectmenu objectdata*) [Function]

Returns an instance of PLOTOBJECT. Coerces objectmenu into a MENU if it is an item list.

The following subtypes of PLOTOBJECT are currently implemented.

pointPLOTOBJECT, curvePLOTOBJECT, polygonPLOTOBJECT, linePLOTOBJECT,
graphPLOTOBJECT, texttPLOTOBJECT, filledrectanglePLOTOBJECT, compound PLOTOBJECT

The functions CREATEPOINT, etc. return an instance of PLOTOBJECT, with the appropriate OBJECTFNS and OBJECTDATA. In order for this to work, some intializations must be done at load time.

The function PLOT.SETUP performs the intializations at LOAD time.

(PLOT.SETUP *opstable*) [Function]

Opstable must be a list of lists of the form:

(

```
(subtypename1  (opname1  function1) (opname2  function2)  ....
(subtypename2  (opname1  function1) (opname2  function2)  ....
 .....
(subtypenamen  (opname1  function1)(opname2  function2) ....
)
```

Creates one instance of PLOTFNS for each subtypename.

In summary, to add a new plot object you need to:

• Determine the data required to describe the new subtype. This may involve declaring a new datatype.

• Write functions similar to CREATEPOINT and PLOTPOINT for the new subtype.

• Write (or borrow) the functions which implement the generic ops described above.

• Invoke MAKEPLOTFNS to create an instance of PLOTFNS for the new plot object subtype, which all objects of that subtype will refer to.

• If continued use of the new plot object is contemplated, PLOT.SETUP should be evoked at load time to effect the proper initializations.

Look at the code for existing plot objects for more details. The point plot object is the simplest example.

PLOTOBJECTS1

By: Tad Hogg (hogg.PA @ Xerox.com)

Uses: PLOT and PLOTOBJECTS

PLOTOBJECTS1 defines additional plot objects for use with PLOT.

NEW PLOTOBJECTS

ERRORPOINT - a point with vertical and/or horizontal error bars.

SAMPLESET - a set of points drawn as line segments to a specified vertical or horizontal line.

FUNCTIONS

The following functions provide an add facility for the new objects. They are similar to the corresponding functions for the standard plot objects, e.g. PLOTPOINT, etc. The allowed forms of the arguments *symbol*, *style*, *menu* and *nodrawflg* are the same as for the standard functions.

(PLOTERRORPOINT *plot position-range label symbol style menu nodrawflg*) [Function]

Position-range is a list of the form (POSITION XRANGE YRANGE). POSITION is the position of the point in world coordinates. XRANGE and YRANGE control the length of the horizontal and vertical error bars respectively. If the range is NIL, no error bars are drawn. If it is a number, it is a distance (in world coords) for the error bar to extend on each side of the point. Finally, if it is a pair of numbers (NegDist . PosDist) it specifies the extent of the error bar in the negative and possitive directions, respectively. Symbol is used to plot the point. Symbol defaults to STAR. Style specifies the style to use for drawing the error bars.

Returns an ERRORPOINT PLOTOBJECT.

(PLOTERRORPOINTS *plot position-ranges labels symbol style menu nodrawflg*) [Function]

As above except that position-ranges is a list of POSITION-RANGEs as described above and labels may also be a list. Reasonable things happen if positions and labels are of unequal length.

Returns a list of ERRORPOINT PLOTOBJECT's.

(PLOTSAMPLESET *plot positions constant vertical? side label style menu nodrawflg*) [Function]

The list of POSITION's defines a number of sample points. Constant specifies the location of a vertical or horizontal line, depending on whether vertical? is non-NIL. Line segments are drawn from the sample points to this line. Side determines which points are actually included. If side is NIL, only those points whose coord is greater than constant will be drawn (i.e. points above or to the right of the line). If side is T, only those with coord less than constant will be drawn. Otherwise, all points will be included. Style specifies the style to use for drawing the line segments.

Returns a SAMPLESET PLOTOBJECT.

All plot objects may be created independently of the previous functions. This is useful if it is desired to create a plot object without entering it on a PLOT's display list. The following functions create and return the new plot objects.

(CREATEERRORPOINT *position-range label symbol style menu*) [Function]

Returns an ERRORPOINT PLOTOBJECT.

(CREATESAMPLESET *positions constant vertical? side label style menu*) [Function]

Returns a SAMPLESET PLOTOBJECT.

In addition there are a number of functions to aid in creating position-ranges used with the error point plot objects:

(MAKE-POSITION-RANGE *position xrange yrange*) [Function]

Returns a position-range suitable for use for specifying an error point. The arguments are as described above for PLOTERRORPOINT.

(LOG-ERROR-RANGE *position-range axis base*) [Function]

Returns a position-range corresponding to *position-range* converted to a log scale. *base* is the log base to use (defaults to 10) and *axis* specifies which axis to convert: :X or :Y for a specific axis, NIL for both. Note that the position, with its error bars must be positive in order to be converted to a log scale.

(LOG-ERROR-RANGE-LIST *position-ranges axis base*) [Function]

Converts a list of *position-ranges* to log scale.

PLOTOBJECTS2

By: Tad Hogg (hogg.PA @ Xerox.com)

Uses: PLOT and PLOTOBJECTS

PLOTOBJECTS2 defines an additional plot object for use with PLOT.

NEW PLOTOBJECT

FILLEDPOLYGON - a polygon with optional shading in its interior.

FUNCTIONS

The following functions provide an add facility for the new object. They are similar to the corresponding functions for the standard plot objects, e.g. PLOTPOINT, etc. The allowed forms of the arguments *texture*, *style*, *menu* and *nodrawflg* are the same as for the standard functions.

(PLOTFILLEDPOLYGON *plot positions label style texture menu nodrawflg*) [Function]

The points in *positions* define a closed polygon. The polygon is filled with *texture*, and the other arguments are the same as for PLOTPOLYGON. If the linewidth specified by *style* is 0, the polygon perimeter will not be drawn.

Returns a FILLEDPOLYGON PLOTOBJECT.

All plot objects may be created independently of the previous functions. This is useful if it is desired to create a plot object without entering it on a PLOT's display list. The following creates and returns the new plot object.

(CREATEFILLEDPOLYGON *positions label style texture menu*) [Function]

Returns aa FILLEDPOLYGON PLOTOBJECT.

The actual drawing is done with

(CLILPPED.FILLPOLYGON *clippingregion points texture stream operation windnumber draw? width drawoperation color dashing*) [Function]

which effectively does both of (FILLPOLYGON *points texture stream operation windnumber*) and (if *draw?* is non-NIL)a series of (DRAWLINE *x1 y1 x2 y2 width drawoperation stream color dashing*) for each edge of the polygon except they are clipped against *clippingregion* in stream coordinates.

PREEMPTIVE

By: Larry Masinter (Masinter.pa@Xerox.com)

This module turns on pre-emptive process scheduling. Using IL:\\PERIODIC.INTERRUPT, it forces a block in whatever process is running.

(IL:PREEMPTIVE &OPTIONAL *STATE*) [Function]

The function PREEMPTIVE turns preemptive process scheduling on and off. (IL:PREEMPTIVE ':ON) turns it on, (IL:PREEMPTIVE ':OFF) turns it off. (IL:PREEMPTIVE) with no argument returns the current state with no change.

WARNING WARNING WARNING WARNING DANGER DANGER DANGER DANGER

PREEMPTIVE is dangerous. Many places in the system do not have monitor locks and other mechanisms to prevent one process from overwriting the data of another in the face of preemptive interrupts. (Most do, of course.)

I've run with preemptive scheduling turned on for weeks, and about once a day, my screen gets trashed, windows and menus overwritten, etc. This version of PREEMPTIVE is a little more conservative than previous versions, e.g., it checks to see if the system is running in the MENU code and doesn't do a process switch. However:

USE AT YOUR OWN RISK. CAUTION CAUTION.

NOTE: Using SPY turns preemptive scheduling OFF.

PRESSFROMNS

By: Tad Hogg (Hogg.pa@Xerox.com)

INTRODUCTION

This module is a patch to allow Press printers to print NS characters by translating them to appropriate Press fonts. Before loading this file, make sure there are no open press streams (i.e. no hardcopy in progress to a Press printer).

CONTROLLING CHARACTER SET TRANSLATIONS

The translations are controlled by a number of variables and functions described below. These variables can be modified to provide additional or different translations.

Global character set translations

NSTOASCIITRANSLATIONS [Variable]

an ASSOC list whose elements have the form (charset translationArrayName). This specifies which translation array is to be used when translating the specified character set.

Example: `((0 ASCIIFROM0ARRAY) (38 ASCIIFROM38ARRAY))` specifies that ASCIIFROM38ARRAY is bound to the array to be used for translating charset 38.

The translationArrayName is bound to an array whose index ranges from 0 to 255. Each element of the array specifies the translation to use for the corresponding charcode in this charset.

Translations are of one of the following forms:

1. NIL -- no translation specified which will use the font as is for charset 0 and otherwise print a black box to indicate the NS character could not be translated

2. an integer in the range 0 to 255 which indicates that this value is to be used as the translated charcode, but that no font translation is required [This is mainly useful for converting NS to ASCII in charset 0.]

3. a two element translation list of the form (fontFamily charcode) which indicates that this character should be translated to charcode in a font whose family is fontFamily. Note that charcode should be in the range 0 to 255. fontFamily can also be a font descriptor or a font specification list (e.g. (Gacha 10)) of a form acceptable to FONTCREATE.

PRESSFONTFAMILIES [Variable]

a list whose elements are of the form (FAMILY . specialTranslations). The optional list specialTranslations specifies translations to use for Press fonts in charset 0. Each element is (charcode translation) which specifies that translation is to be used for charcode in this family.

Example: `((GACHA (50 (HELVETICA 40)) (51 (TIMESROMAN 45))) (TIMESROMAN) (SYMBOL))`

Note: these translations are cached in the font descriptor when PRESS fonts are created so any changes to these variables will not change the translations in previously created fonts.

Translations in individual fonts

The following two functions provided detailed control over the translations used in individual fonts.

(GETCHARPRESSTRANSLATION *CHARCODE FONT*) [Function]

returns the translation (a two element list) used for CHARCODE in FONT

(PUTCHARPRESSTRANSLATION *CHARCODE FONT NEWTRANSLATION*) [Function]

sets the translation to be used for CHARCODE in FONT. NEWTRANSLATION should be a translation in one of the forms described above.

Additional functions

(PRESS.NSARRAY *CHARSET FAMILY ASCIIARRAY*) [Function]

This function returns a suitable translation array built as the inverse of a translation array from Press to NS characters. Such arrays are used to print Press fonts on Interpress printers and are listed in the variable ASCIITONSTRANSLATIONS. CHARSET is the character set for which to create a translation. FAMILY is the press font family for which ASCIIARRAY is the translation to NS characters. If ASCIIARRAY is not specified, the function looks through all arrays included on ASCIITONSTRANSLATIONS to fill in the translation array.he following two functions provided detailed control over the translations used in individual fonts.

CONTROLLING PRESS FONT COERCIONS

There is also a mechanism for determining which press font is actually used for the translation. For example, an NS character in the Modern 8 font might translate to an ASCII character in Symbol 8. If this font does not exist on the printer, a (generally incorrect) font change will be done by the printer. The following procedure changes the actual font used, e.g. to Symbol 10 in this example.

The coercion is controlled by a coercion list which is an alist indexed by device. The font coercion scans down the element on the list for the requested device (e.g. PRESS) looking for the first entry that matches the user request. If a match is found, then the entry tells how to construct an appropriate new name from the requested specification. Fields of the newname not specified in the entry are simply copied over.

FONTCOERCIONS [Variable]

This list allows the user to coerce fonts that he knows don’t exist on the printer even tho the fonts-widths files doesn’t indicate that (e.g. the desired size doesn’t exist). FONTCOERCIONS is initialized simply to take all SYMBOL fonts of size less than 10 into 10, and size greater than 12 into 12.

MISSINGFONTCOERCIONS [Variable]

If the initial coerced (or uncoerced) lookup fails, then MISSINGFONTCOERCIONS is used. This takes MODERN into HELVETICA etc--the standard press coercions.

The procedure for determining whether a user request matches a coercion entry is straightforward. If the match-part of the coercion entry is an atomic family name, it matches if it is eq to the requested family. Otherwise, the match-part must be a list of family, size, face in standard fontname order. If a component is NIL or missing, then it is assumed to match. The only funniness is in size matching. The size component can be NIL (matches anything), a particular size (EQ matches), or a list of the form (<

n) or (> n) where n is a size number. The first matches any requested size less than n, and the second matches any requested size greater than n.

FONTCOERCIONS for PRESS starts out as

```
((SYMBOL (< 10) ) (SYMBOL 10)) (SYMBOL (> 12))(SYMBOL 12))
```

MISSINGFONTCOERCIONS for PRESS starts out as

```
((MODERN HELVETICA)(CLASSIC TIMESROMAN) etc.)
```

PRETTYFILEINDEX

By: Bill van Melle (vanMelle.PA@Xerox.com)

INTRODUCTION

PRETTYFILEINDEX is a program for generating indexed listings for Lisp source files. PRETTYFILEINDEX operates by reading expressions from the file and reprettyprinting them to the output image stream, building up an index of the objects as it goes. The index is partitioned by type (e.g. FUNCTIONS, VARIABLES, MACROS, etc.); within each type, the objects are listed alphabetically by name along with the page number(s) on which their definitions appear in the listing.

PRETTYFILEINDEX also modifies the Exec's and the FileBrowser's SEE command to prettyprint the file being viewed, if it is a Lisp source file. It also modifies the PF and PF* commands to prettyprint the requested function body. Together, these features mean you can use the NEW & FAST options to MAKEFILE to speed up file creation without sacrificing the ability to get pretty listings or see the files prettily inside Lisp.

PRETTYFILEINDEX performs some additional niceties in the listing: it prints bitmaps by "displaying" them, rather than dumping their bits; it translates underscore to left arrow (for the benefit of Interlisp listings); it prints quote and backquote in a font in which they are clearly distinguishable; and it suppresses some of the "noise" in source files, such as the filemap.

The module also contains a function MULTIFILEINDEX that can be used to generate a merged index of items from a whole set of files being listed.

PRETTYFILEINDEX subsumes, and is incompatible with, the modules SINGLEFILEINDEX and PP-CODE-FILE. You can, however, load PRETTYFILEINDEX on top of either one, and it will successfully wrest control of LISTFILES from them. PRETTYFILEINDEX has several advantages over SINGLEFILEINDEX: the prettyprinter has fine control over positioning of the output stream, so things that are supposed to line up do, despite font changes and variable-width fonts; the entire page is used, rather than sacrificing the bottom quarter or so due to lack of control over page breaks; and the use of an image stream allows bitmaps to be rendered directly.

USING PRETTYFILEINDEX

For ordinary use, just load PRETTYFILEINDEX.LCOM. This redefines LISTFILES1 so that calling LISTFILES or using the File Browser's Hardcopy command invokes PRETTYFILEINDEX if the file is a Lisp source file. The listing is created by default in a single background process that handles all LISTFILES requests. The file being indexed needn't be loaded, or even noticed (in the File Manager sense) as long as the file's commands don't require customized prettyprinting defined by the file itself. The index is printed at the end of the listing; you are expected to manually transpose the index to the front of the collection of paper that emerges from the printer.

PRETTYFILEINDEX normally assumes that you are printing one-sided listings. However, if your global default is for two-sided (currently this means that EMPRESS#SIDES = 2) or you specified two-sided in the options you passed to LISTFILES, it will prepare the output as if for two-sided listing. For example, from an Interlisp exec,

```
(LISTFILES (SERVER "Perfector:" %#SIDES 2) FOOBAR)
```

causes the file FOOBAR to be listed two-sided on the print server Perfector: (the `%` is the Interlisp reader's escape character, needed to quote the special character #; in an XCL exec the escape character is \, and from other packages you also have to qualify the symbols `LISTFILES`, `SERVER` and `#SIDES` with the package prefix `IL:`).

For two-sided listings, the margins are symmetric, instead of being shifted a bit to the right, page numbers appear on the outside edge of the page, and a blank page is inserted at the end of the listing if necessary to ensure that the index starts on an odd page (and hence is transposable to the front).

PRETTYFILEINDEX prettyprints the file's contents and prints indexed names using the package and read table specified in the file's reader environment, which appears at the beginning of the file. It assumes, as does most of the file manager, that the reader environment is sufficient to read any expression on the file. If you have violated this assumption, for example, by referring in the file to a symbol in another package that is defined on a file that is indirectly loaded by the file somewhere in its coms, you will probably need to LOADFROM the file before you can list it.

INDEXING MULTIPLE FILES

Ordinarily, you list files and get one index per file. If a module is made up of several files, you may want a master index of the whole set of files, so that you don't have to remember which file contains a function, macro, etc. that you are looking up. This job is handled by MULTIFILEINDEX:

(MULTIFILEINDEX *files printoptions*) [Function]

> This function lists each of the files in the list *files* using PRETTYFILEINDEX and then produces a master index by merging all the individual indices. The master index is appended to the output of the last file listed. The argument *files* can be a list of file names and/or file patterns, such as "{FS:}<Carstairs>RED*", or a single such pattern. In the pattern, unless explicitly specified, the extension defaults to null and the version to "highest". The argument *printoptions* is a property-list of options, the same as the *printoptions* argument to SEND.FILE.TO.PRINTER or PRETTYFILEINDEX, with the addition of some options recognized by MULTIFILEINDEX, described further below.

As each file is listed, its pages are numbered with an ordinal file number plus the page number within the file; e.g., in the first file the pages are numbered 1-1, 1-2, ..., in the second file 2-1, 2-2, etc. The master index then refers to page numbers in this form, although each individual file's own index shows only the file-relative page numbers. Alternatively, you can tell MULTIFILEINDEX to number all the pages consecutively, rather than using "part numbers", by giving the option `:CONSECUTIVE`, value `T` in *printoptions*.

In the event that some files in the set have different reader environments, the master index is printed in the environment used by the majority of the files. More specifically, MULTIFILEINDEX independently chooses the package used by the majority of the files and the readtable used by the majority; in the case of a tie, the file later in the set wins. If this default is not adequate, you can specify the environment yourself by giving the `:ENVIRONMENT` option. The value should either be a reader environment object, such as produced by `MAKE-READER-ENVIRONMENT`, or a property list of the form used by the `MAKEFILE-ENVIRONMENT` property.

For example,

```
(MULTIFILEINDEX "<Barney>Rub*"
          '(:CONSECUTIVE T
            :ENVIRONMENT (:PACKAGE "JABBA" :READTABLE "XCL")))
```

would list each of the files matching "<Barney>Rub*.;", numbering the pages consecutively from the first file through the last, and printing the master index with respect to the package JABBA and read table XCL.

INCREMENTALLY REPRINTING MULTIPLE FILES

If you have used MULTIFILEINDEX to list a group of files, and later one of the files changes, or maybe the printer just ate part of your listing, you might want to update your listing without reprinting the entire set of files. You have two options.

(1) You can have PRETTYFILEINDEX reprint the one file that changed (or was eaten). Specify the print option `:PART` *n* to have it treat the single file as the *n*th part of a multiple listing, or the option `:FIRSTPAGE` *n* to have it start numbering the pages at *n* instead of 1 (for the case where you used the `:CONSECUTIVE` option to MULTFILEINDEX). For example,

```
(LISTFILES (:PART 3) "<Barney>Rubric")
```

would reprint `<Barney>Rubric` as the third file in a group. Of course, this doesn't reprint the master index, but it only has to process the one file, which may be adequate for your needs if things didn't move around too much.

(2) You can have MULTIFILEINDEX process the entire set of files again, but only print some of them. You specify this by parenthesizing the files you don't want printed. That is, each element of the *files* argument to MULTIFILEINDEX is a file name or a list of file name(s); those files inside sublists are processed but not printed. You cannot specify patterns. The master index is listed after the last file, as usual, except that if the last file was in a sublist, and hence not printed, the master index will appear as a separate listing. Calling MULTIFILEINDEX in this manner is nearly as computationally expensive as calling it to list the whole set for real (it omits only the transportation to the printer), but it does save paper and printer time.

LISTING COMMON LISP FILES

Ordinarily, PRETTYFILEINDEX only processes files produced by the Lisp File Manager; it passes all others off to the default hardcopy routines. However, you can tell it to process a plain Common Lisp text file by passing the print option `:COMMON`; e.g.,

```
(LISTFILES (:COMMON T) "conjugate.lisp")
```

PRETTYFILEINDEX still processes the file by reading and prettyprinting, just as for Lisp files. It starts in the default Common Lisp reading environment (package USER and read table LISP), and evaluates top-level package expressions, such as `in-package` and `import`, in order to continue reading correctly. The index is printed in whatever the environment was at the end of the file.

Of course, this is of fairly limited utility, as all read-time conditional syntax is lost: comments, #+, #o, etc. The one exception is that top-level semi-colon comments are preserved—they are copied to the output directly, rather than being read.

Customizing PRETTYFILEINDEX

The remainder of this document describes various ways in which PRETTYFILEINDEX can be customized.

HOW TO SPECIFY INDEXING TYPES

Initially, PRETTYFILEINDEX knows about most of the standard file manager types. In addition, it handles all the types defined by DEFDEFINER. For definers with a :NAME option, it assumes that the function is free of side effects. PRETTYFILEINDEX also notices (but does not evaluate) DEFDEFINERs that appear on the file it is currently indexing, which should appear before any instances of the type so defined in order for correct indexing to occur. Of course, it can't know about definer types that are defined on some other file unless you load it.

You can augment the set of indexing types, or override the default handling of definers, by adding elements to the following variable:

`*PFI-TYPES*` [Variable]

A list of entries describing types to be indexed and a way of testing whether an expression on the file is of the desired type. Each entry is a list of up to 4 elements of the form (*type dumpfn namefn ambiguous*), the first two of which are required:

type The name of the type, e.g., `MACRO`. This name will appear as the name of the index for this type, e.g., "MACRO INDEX". *type* is usually the name of a file package type, though it need not be. It must be a symbol.

dumpfn The name of the function that appears as the CAR of the form that defines objects of type *type* on the file, or a list of such names. E.g., for type `TEMPLATE` it is `SETTEMPLATE`; for type `VARIABLES` it is `(RPAQ RPAQQ RPAQ? ADDTOVAR)`.

namefn A function that tests whether the expression that starts with *dumpfn* really is of the desired type, and returns the name of the object defined in the expression. The function takes as arguments (*expr entry*), where *expr* is the expression whose CAR matched the entry. The *testfn* should return one of the following:

`NIL` the expression is not of the desired type.

name the expression defines a single object of this name and of the type given in the entry.

a list the value is either a single list or a list of lists, each of the form (*type . names*), meaning that the expression defines each of the names as having the specified type.

If the *namefn* is `NIL` or omitted, the name of the object is obtained from the second element of the expression. If that element is a list, the name is taken to be its CAR, or its CADR if the element is a quoted atom.

ambiguous True if the expression is ambiguous, in the sense that even if *namefn* returns a non-NIL value, it is possible for this expression to also satisfy other entries in `*PFI-TYPES*`. E.g., the expression `(RPAQ --)` is ambiguous, because it could define either a variable or a constant. If *ambiguous* is true, you usually want a corresponding entry on `*PFI-FILTERS*` (below).

PFI-PROPERTIES [Variable]

A list used by the default handler for the PUTPROPS form. It associates property names with a type (something more specific than the type PROPERTY) under which objects having this property should be indexed. Each element is of the form (*propname type*). If *type* is NIL or omitted, then objects having this property are ignored. In addition, the default PUTPROPS handler treats all elements of the list MACROPROPS as implying type MACRO.

The initial value of *PFI-PROPERTIES* is

```
((COPYRIGHT)
 (READVICE ADVICE)),
```

meaning that the COPYRIGHT property should be ignored, and the READVICE property implies that the object should be indexed as type ADVICE.

PFI-FILTERS [Variable]

A list describing potential index entries that should be filtered out of the final index. Each element of *PFI-FILTERS* is a list (*type filterfn*), where *type* is one of the types in *PFI-TYPES* and *filterfn* is a function of one argument, an index entry. If *filterfn* returns true, then the index entry is discarded. An index entry is of the form (*name . pagenumbers*). For convenience, an element of *PFI-FILTERS* can also take the form (*type . subtype*), meaning that if an object is already indexed as a *subtype* then it should not also be indexed as a *type*.

The initial value of *PFI-FILTERS* is

```
((VARIABLES . CONSTANTS)),
```

meaning that "variables" that successfully index as constants should not also be listed in the VARIABLES index. This extra pass is needed because the CONSTANTS File Manager command causes expressions of the form (RPAQ *var value*) to be dumped on the file, and at the time this expression is read, it is not known whether there will later on appear a CONSTANTS form for the same variable.

Filter functions may want to call the following function:

(PFI.LOOKUP.NAME *name type*) [Function]

Looks up *name* in the index being built for type *type*. If it finds an entry, it returns it. Index entries are of the form (*name . pagenumbers*). It is permissible for a filter function as a side effect to destructively change another index entry by adding page numbers to it. You might want to do so, for example, in the case where there is a kind of object that dumps two expressions on a file, each of which is a different type (according to *PFI-TYPES*), but you want both occurrences indexed as a single type.

MORE EXPLICIT EXPRESSION HANDLING

The functions and variables described below allow you to completely control how certain expressions in the input file are handled. You can use these hooks to perform custom prettyprinting, to suppress the printing of some expressions, or to perform indexing more complex than that supported by *PFI-TYPES*.

PFI-HANDLERS [Variable]

An association list specifying explicit "handlers" for expressions that appear on the input file. Each element is a pair (*car-of-form . handler*), where *handler* is a function of one argument, an expression read from the file whose first element is *car-of-form*. The handler is completely in charge of indexing the expression and/or printing it to *STANDARD-OUTPUT*. Unless the handler chooses to suppress the printing altogether, it is expected to print at least one blank line first, so that expressions are attractively separated in the listing (see PFI.MAYBE.NEW.PAGE).

PFI-PREVIEWERS [Variable]

This list is used when PRETTYFILEINDEX is used by the SEE command. During the SEE command, real-time performance is important, so it is undesirable to have long delays while reading a very large expression. For example, all the functions in an Interlisp FNS command appear on the file inside a single DEFINEQ expression. If handled in the obvious way, the user would have to wait for the entire expression to be read before any output appeared. A previewer has the opportunity to read the expression in pieces and prettyprint it as it goes.

Each element of *PFI-PREVIEWERS* is a pair (*car-of-form . previewer*), where *previewer* is a function of one argument, the *car-of-form*. The previewer is called when PRETTYFILEINDEX encounters an expression of the form "(*car-of-form* " on the file. Its job is to read expressions from *STANDARD-INPUT* (currently positioned after the car of form) until it encounters the closing right parenthesis, which it should consume, and prettyprint the elements appropriately to *STANDARD-OUTPUT*. *PFI-PREVIEWERS* is used only from the SEE command, so indexing is not necessary (but also not harmful, other than to waste some time).

If an expression does not have a previewer, PRETTYFILEINDEX reads the reset of the expression itself and handles it normally, i.e., performs (PFI.HANDLE.EXPR (CONS *car-of-form* (CL:READ-DELIMITED-LIST #\))).

(PFI.DEFAULT.HANDLER *expr*) [Function]

This is the function PRETTYFILEINDEX uses to process expressions that have no explicit handler. It indexes the expression according to *PFI-TYPES* and then prettyprints the expression. You can call this function from your handler if you decide you have an expression you didn't want to handle specially.

(PFI.HANDLE.EXPR *expr*) [Function]

Performs PRETTYFILEINDEX's normal handling of the expression *expr*, including looking on *PFI-HANDLERS*. Handlers and previewers of forms that encapsulate arbitrary expressions, such as DECLARE:, typically call this to process subexpressions.

(PFI.ADD.TO.INDEX *name type/entry*) [Function]

Adds an entry to the index for *type/entry* specifying that *name* occurs on the current page. *type/entry* is either a type or an entry from *PFI-TYPES* from which the type will be extracted.

(PFI.PRETTYPRINT *expr name formflg*) [Function]

Prettyprints *expr*. Optional *name* is the name of the object being printed; if a page crossing occurs in the middle of the prettyprinting, this name will be displayed in the page header. If *formflg* is true, print the expression as code; otherwise as data.

(PFI.MAYBE.NEW.PAGE *expr minlines*) [Function]

Starts a new page if the listing is currently near the bottom of the page and *expr* won't fit, else performs a single (TERPRI). If *minlines* is specified, it is an explicit estimate of how much space the expression will require, in which case *expr* can be NIL; otherwise, the function estimates the size. Handlers should call this *before* calling PFI.ADD.TO.INDEX, so that the page number in the index is correct. The typical handler calls PFI.MAYBE.NEW.PAGE, then PFI.ADD.TO.INDEX, then prints the expression, possibly via PFI.PRETTYPRINT.

OTHER VARIABLES

PFI-INDEX-ORDER [Variable]

A list of types (as in *PFI-TYPES*) in the order in which the various types should appear in the index. Types not in this list are printed in an order of the program's choosing, currently a "best fit" algorithm (print the largest type index that will fit on the page). The initial value is (FUNCTIONS), meaning that the function index will appear first, with no constraints on the order of other types.

PFI-PRINTOPTIONS [Variable]

A plist of print options that PRETTYFILEINDEX appends to the list of print options passed to LISTFILES, thus supplying some printing defaults. The initial value is (REGION (72 54 504 702)), which on standard letter size paper in portrait mode results in left, bottom, top, and right margins of 1", ¾", ½" and ½", respectively. If the print options passed to LISTFILES call for a two-sided listing, the default region is shifted ¼" to the left. If the print options specify LANDSCAPE mode, the default region is ignored. Any REGION option specified in *PFI-PRINTOPTIONS* must be in points; it is scaled appropriately to the actual hardcopy device being used.

PFI-MAX-WASTED-LINES [Variable]

If an expression looks like it won't fit on the current page and there are no more than this many lines remaining on the page, PRETTYFILEINDEX starts a new page before printing the expression. A floating-point value indicates a fraction of the page; an integer indicates an absolute number of lines. The initial value is 12.

PFI-CHARACTER-TRANSLATIONS [Variable]

A list specifying how certain characters should be rendered on the output stream. This is used to get around the poor rendering of certain characters in the default font. Each element is of the form (*imagetype . charpairs*), where *imagetype* is the type of image stream being printed to and each element of *charpairs* is an alist whose elements are of the form (*sourcecode destcode . looks-plist*), specifying the character code to use on the destination image stream for a specified character code in the input stream. If *looks-plist* is non-NIL, *destcode* is printed in a font obtained by applying FONTCOPY to the current font and *looks-plist*.

The initial value is

```
((INTERPRESS (95 172)
             (96 169 FAMILY CLASSIC)
             (39 185 FAMILY CLASSIC)))
```

meaning if the output stream is an Interpress stream the lister should turn character 95 (underscore) into 172 (left arrow), backquote into left single quote in the Classic font (of the same size and weight), and single quote into right single quote in Classic.

PRINT-PRETTY-FROM-FILES [Variable]

If true, the SEE (in the Exec and Filebrowser), PF and PF* commands attempt to prettyprint to the display, rather than copying the file as it is currently formatted. The initial value is T.

PRINT-PRETTY-BITMAPS [Variable]

If true, then when *PRINT-ARRAY* is true and a bitmap is to be printed to an image stream, the bitmap itself is displayed as an image on the stream, rather than as the machine-readable representation of its bits (of the form #*(16 16)H@@@L...). This variable has no effect on printing to files, such as in MAKEFILE, nor on PRETTYFILEINDEX, which binds it true; thus, changing the value mainly affects the display. The initial value is T.

PFI-DONT-SPAWN [Variable]

If NIL, LISTFILES arranges for a separate process to do the hardcopying (whether using PRETTYFILEINDEX or not) and returns immediately; if T, it makes the listing directly, not returning until it is finished. The initial value is NIL.

LISTING ELSEWHERE THAN THE PRINTER

Ordinarily, you call LISTFILES (or uses the File Browser) to create listings. However, you can also call PRETTYFILEINDEX directly if you want to direct the output elsewhere, such as to an Interpress file:

(PRETTYFILEINDEX *filename printoptions outstream dontindex*) [Function]

Lists *filename*, the name of a Lisp source file or a stream open for input on such a file, printing it and its index to *outstream*. *outstream* is either an open image stream, or NIL, in which case the output goes to (OPENIMAGESTREAM) and the stream is closed afterwards, which results in it being sent to the default printer. If *filename* or *outstream* is open on entry, it is left open on exit. *printoptions* is a plist of options of interest to either LISTFILES or OPENIMAGESTREAM. If *dontindex* is true, no index is produced; this argument is used by the SEE command.

If the file is not a File manager file, PRETTYFILEINDEX takes no action and returns NIL; otherwise, it returns the full file name. However, if *filename* is an open stream, then PRETTYFILEINDEX copies the remainder of the stream to *outstream* (which must be given) using PFCOPYBYTES, and returns the full file name. This is so that the stream does not need to be backed up after discovering that the file is not a File Manager file, an operation not possible for a sequential-access stream.

LIMITATIONS

PRETTYFILEINDEX assumes that the default font, which is used to print the index, is fixed-width.

PRETTYFILEINDEX uses the regular Interlisp prettyprinter. This means that if you have File Manager commands that produce their output in a customized way, e.g., by printing inside the E command, then the output will look different between MAKEFILE and PRETTYFILEINDEX. You can usually remedy this by supplying PRETTYPRINTMACROS for the types of expressions your command dumps (which may also let you replace the E with a simpler P command), or by defining handlers for the expressions (see *PFI-HANDLERS*). PRETTYFILEINDEX already supplies PRETTYPRINTMACROS for most of the customized printing done by the current File Manager: RPAQ, RPAQQ, RPAQ?, ADDTOVAR, PUTPROPS and COURIERPROGRAM.

With the exception of noticing the reader environment and DEFDEFINER expressions, PRETTYFILEINDEX does not interpret the contents of the file. If your file depends on itself for proper prettyprinting or indexing, you need to LOAD (or possibly just LOADFROM) the file first.

PRINTERMENU

By: ISLWS (Bloomberg.pa@Xerox.com)

DESCRIPTION

Creates a menu which displays all printers in the global list DEFAULTPRINTINGHOST, allows selection of a default printer, and permits addition and deletion of printers from DEFAULTPRINTINGHOST. Printers are displayed in the same order as they appear in DEFAULTPRINTINGHOST. Selecting an item from the menu will highlight by inversion and move it to the top of the menu, thus becoming the default printer. Selection in the title bar of the menu with the left or middle button will allow you to add or to delete a printer from the menu.

An auxiliary process, PRINTERMENU.WATCH, monitors the value of DEFAULTPRINTINGHOST and will update the menu if this variable is changed. If PRINTERMENU.WATCH is killed, the menu will be grayed out to indicate that it may no longer be valid. Clicking left or middle buttons inside the menu will restart PRINTERMENU.WATCH and update the menu.

To use:

Load the module with:

(LOAD 'PRINTERMENU.LCOM)

Start it with:

(PRINTERMENU)

User Switches:

1. Set DEFAULTPRINTINGHOST to contain all printers from which you wish to select.
2. Prior to calling the function (PRINTERMENU), the global variable PRINTERMENU.POSITION can be set to the position, in screen coordinates, where you want the menu to appear. If not set, you will be prompted for a position for the menu window.

PROGRAMCHAT

By: ISLWS (bloomberg.pa@xerox.com)

DESCRIPTION

PROGRAMCHAT is a Lisp function that invokes a windowless Chat process to execute a single command line on a remote host. PROGRAMCHAT requests a *login* if one has not been made recently to the remote host. After execution of the command, a normal *logout* is performed, and the Chat connection is closed.

PROGRAMCHAT was written by Eric Schoen to allow initiation of remote computation from Lisp workstations. It works with both VMS and Unix operating systems on the remote host.

To use:

Load the module with:

```
(LOAD 'PROGRAMCHAT.LCOM)
```

Invoke the functon with:

(PROGRAMCHAT *hostname commandString windowFlg*) [Function]

where

hostname is the network name of the remote host,

commandString is a string which is the exact format of the command to be run from the command line interpreter of a VAX/VMS host (or from the shell of a VAX/Unix host), and

windowFlg is a variable that, when T, opens a window and displays a log of data transferred between the PROGRAMCHAT process and the remote host. PROGRAMCHAT is normally invoked with *windowFlg* = NIL.

Warnings:

1. When loaded, PROGRAMCHAT resets the variable NETWORKLOGINFO.
2. PROGRAMCHAT provides no error handling. If the connection to the remote host is broken, no error message is returned.

PROMPTREMINDERS

To be periodically reminded of things

By: JonL White

Revised by: Larry Masinter (Masinter.pa@Xerox.com), subsequently by Becky Burwell (Burwell.PA@Xerox.COM)

INTRODUCTION

PROMPTREMINDERS implements a facility which schedules events to be performed, or messages to be flashed in a prompt window. Events can be periodic or once-only. The showing of a message in a prompt window has the extra facility of flashing a message, and stopping only when there has been a recent response (mouse or keyboard movement) from the user.

If the MESSAGE given for the reminder (see description of the function SETREMINDER below) is a listp, then when the reminder "goes off", that listp will be EVAL'd rather than any of the "winking", "flashing", or "hassling" mentioned above.

The global variable REMINDERSTREAM holds the stream where the message is to be displayed; if not set by the user, it defaults to PROMPTWINDOW. After the message has been displayed, the window (if indeed REMINDERSTREAM holds a window) will be closed, depending on the value of CLOSEREMINDERSTREAMFLG.

REMINDERS is a file package type, so that they may be easily saved on files, and so that the general typed-definition facilities may be used. On any file which uses the REMINDERS filepkgcom, it is advisable to precede this command with a command

```
(FILES (SYSLOAD COMPILED FROM LISPUSERS) PROMPTREMINDERS)
```

since this package is not in the initial Lisp loadup. When initially defining a reminder, it is preferable for the user to call SETREMINDER rather than PUTDEF; but HASDEF is the accepted way to ask if some name currently defines a "reminder", and DELDEF is the accepted way to cancel an existing "reminder".

EXAMPLES

```
(SETREMINDER NIL (ITIMES 30 60) "Have you done a CLEANUP recently?")
```

the user wants to be reminded every 30 minutes that he ought to save his work

```
(SETREMINDER 'WOOF NIL "Call home about dinner plans."
                  "8-Jan-83 4:00PM")
```

he merely wants to be told once, at precisely 4:00PM to call home

```
(SETREMINDER NIL 600
             '(PROGN (AND (FIND.PROCESS 'LISTFILES) (add FREQ 1))
                      (add TOTAL 1)))
```

checks every 10 minutes to see if there is a process called LISTFILES

FUNCTIONS

(SETREMINDER NAME *PERIOD MESSAGE INITIALDELAY EXPIRATION*) [Function]

This will create and install a "reminder" with the name NAME (NIL given for a name will be replaced by a gensym), which will be executed every PERIOD number of seconds by winking the string MESSAGE into the prompt window; if MESSAGE is null, then NAME is winked; if MESSAGE is a listp, then it is EVAL'd and no "winking" takes place. "Winking" means alternately printing the message and clearing the window in which it was printed, at a rate designed to attract the eye's attention.

The first such execution will occur at PERIOD seconds after the call to SETREMINDER unless INITIALDELAY is non-NIL, in which case that time will be used; a numeric value for INITIALDELAY is interpreted as an offset in seconds from the time of the call to SETREMINDER, and a stringp value is an absolute date/time string.

If PERIOD is null, then the reminder is to be run precisely once. If EXPIRATION is non-null, then a fixp means that that number of seconds after the first execution, the timer will be deleted; a stringp means a precise date/time at which to delete the timer.

Optional 6th and 7th arguments -- called REMINDINGDURATION and WINKINGDURATION -- permit one to vary the amount of time spent in one cycle of the wink/flash loop, and the amount of time spent winking before initiating a "flash". The attention-attracting action will continue for REMINDINGDURATION seconds (default: the value of the global variable DEFAULT.REMINDER.DURATION which is initialized to 60), or until some keyboard action takes place.

Type-ahead does not release the winking. In case the user fails to notice the winking, then every WINKINGDURATION seconds (default: the value of the global variable DEFAULT.REMINDER.WINKINGDURATION which is initialized to 10) during the "reminding", the whole display videocolor will be wagged back and forth a few times, which effects a most obnoxious stimulus.

SETREMINDER returns the name (note above when NIL is supplied for the name).

(ACTIVEREMINDERNAMES) [Function]

ACTIVEREMINDERNAMES returns the list of active reminders.

(REMINDER.NEXTREMINDDATE *NAME DATE*) [Function]

REMINDER.NEXTREMINDDATE returns (and optionally sets) the date&time (in DATE format) at which the reminder is next to be executed.

(REMINDER.EXPIRATIONDATE *NAME DATE*) [Function]

REMINDER.EXPIRATIONDATE returns (and optionally sets) the date&time (in DATE format) at which the reminder will be automatically deleted.

(REMINDER.PERIOD *NAME SECONDS*) [Function]

REMINDER.PERIOD returns (and optionally sets) the period (in seconds) at which the reminder gets rescheduled.

(SHOWDEF *name* 'REMINDERS) [Function]

will show a reminder in a pretty format, etc.

Proofreader

By: John Maxwell (Maxwell.pa@Xerox.com)

Use in conjunction with Analyzer, SpellingArray

INTRODUCTION

The Proofreader interactively looks for and corrects spelling errors in a given TEdit document. To use it, go to the TEdit menu (click the middle button while in the title bar) and invoke the menu item labeled "Proofread". This will simultaneously attach a special menu to the side of the document and start proofreading from the caret. The proofreader scans the document from the caret location and stops at the first misspelling it finds, highlighting it with a pending delete selection. Successive misspellings can be found by clicking the "Proofread" menu item in the TEdit menu or the "Proofread" menu item in the new side menu.

At this point, you can either correct the misspelled word or skip to the next misspelling. If you are not sure what the correct spelling is, you can get a menu of possible corrections by invoking the "Correct" menu item. Selecting a correction from this menu will cause it to be automatically inserted into the document in place of the misspelling. (Note: The Proofreader occasionally suggests some very bizzare spelling corrections for your misspelled word. Do not be alarmed; this is a known but unavoidable artifact of the heuristic used for checking for misspellings. (see notes at the end)) If the word was erroneously flagged as misspelled, then you can insert the word into your personal word list of acceptable words by invoking the "Insert" menu item. At the end of the editting session you can save the word list on a file with the "StoreWordList" command (see below).

If when the Proofreader is invoked the current selection has more than one word in it, then the Proofreader will only correct the words in that selection. Otherwise, the Proofreader always proofreads the text from the caret to the end of the document.

PROOFREADER SUB-COMMANDS

Under the TEdit Proofread menu item there are a number of sub-commands that the user can invoke. They are:

Proofread

The same as the top level Proofread command. Attaches a special menu to the side of the document and starts proofreading from the caret.

CountWords

Counts the number of words in the current selection. To count the number of words in the entire document, first click "All" in TEdit's expanded menu.

SetProofreader

Gives the user a menu of proofreaders to use for proofreading. If there is only one proofreader, no menu is generated. If the user selects a remote server, a second menu may be generated of proofreaders available on the server.

StoreWordList

Allows the user to save the words that he inserted into the proofreader onto a remote file. The words from an existing version of the file will be read in first and then a new file will be generated consisting of the old file plus the new words. After the file is written, the list of newly inserted words is set to NIL.

LoadWordList

The inverse of **StoreWordList**. If you have a file that you want to load every time you use the proofreader, you can add it to `Proofreader.AutoLoad`, and it will be loaded when the proofreader is first opened. `Proofreader.AutoLoad` can be either a file or a list of files.

AutoCorrect

Sets the variable `Proofreader.AutoCorrect` so that the proofreader *automatically* generates a list of corrections whenever it finds a misspelled word. In the AutoCorrect mode, the proofreader will also automatically scan for the next misspelled word whenever after a correction has been selected. If you want to stop the proofreading process, click outside of the correction menu. If you want to insert the flagged word into your word list, click the menu item labeled "*INSERT*". If you want to continue proofreading without changing the flagged word, select the menu item labeled "*SKIP*".

ManualCorrect

Sets the variable `Proofreader.AutoCorrect` so that the proofreader will not generate a list of corrections unless the user asks for it.

PROOFREADER VARIABLES

There are a couple of variables that the user can set in his init file to change how the proofreader works. They are:

`Proofreader.AutoLoad` [Variable]

A file or list of files to be loaded into the proofreader every time that the proofreader is initialized.

`Proofreader.AutoCorrect` [Variable]

A boolean that determines whether or not a list of corrections is automatically generated whenever the proofreader finds a misspelled word. The default value is NIL.

`Proofreader.AutoDelete` [Variable]

A boolean that determines whether or not to delete the old versions of a word list file when a new one is written out. The default value is T.

`Proofreader.MenuEdge` [Variable]

The side of the window that the proofreader menu appears on (can be either LEFT or RIGHT). The default value is LEFT.

`Proofreader.UserFns` [Variable]

A list of functions to be applied to misspelled words (as strings). If the function returns a non-NIL value, then the word is assumed to be correctly spelled. (If the function ATOMHASH#PROBES is added to `Proofreader.UserFns`, then any word defined as an atom becomes legal. ATOMHASH#PROBES tests whether or not a string is an atom without creating new atoms. If you want a more restricted test (i.e. "anything defined as a procedure is legal") first test that the string exists at an atom before doing MKATOM. Otherwise, the atom space will fill up with misspelled words.)

NOTES

• The proofreader uses a heuristic to determine whether or not a word is in its word list that occasionally will produce false positives. This is most noticeable when the proofreader is generating corrections for a misspelled words. I don't know of any way to eliminate this problem except to use a different algorithm, the fastest of which is at least twice as slow. Hopefully people will find it more of a nuisance than a real problem.

• There is no way to remove words once they have been inserted into the local dictionary. The only way to get rid of a bad word is to reload the dictionary. This can be done by reloading the SpellingArray file. If a bad word gets into one of the remote files, you can edit the file to get rid of it.

ACKNOWLEDEGMENTS

The algorithm used in the Proofreader is based on the algorithm in the Cedar Spelling Tool by Bob Nix. For more information on the implementation, see the section "How it Works" in {Cyan}<CedarChest6.1>Documentation> SpellingToolDoc.tioga.

PSEUDOHOSTS

By Ron Kaplan

This document was created in January 2022.

A pseudohost identifies the root of a file system that exists as a subdirectory of another pre-existing file system. This gives a shorthand way of operating on a file in the subdirectory of a particular project without having to specify in the name of that file the entire path to its location in a larger file system. For example, suppose that the variable `MEDLEYDIR` contains the path from `{DSK}` to the subdirectory that contains all Medley system files (e.g. `{DSK}<Users>kaplan>Local>medley>`). If not connected to that subdirectory, then the file `COREIO`, say, would have to be reference as

`{DSK}<Users>kaplan>Local>medley>sources>COREIO`.

If `MED` is defined as a pseudohost with `MEDLEYDIR` as its prefix, then that file can also be identified more succinctly as `{MED}<sources>COREIO`.

This package implements pseudohost file devices that allow files to be specified and manipulated in this way. The function PSEUDOHOST defines a new pseudohost whose files coincide with the files at the end of a prefix directory path:

`(PSEUDOHOST HOST PREFIX)` [Function]

For the Medley example, executing `(PSEUDOHOST 'MED MEDLEYDIR)` will set up MED as a (pseudo) host name that can be used to reference Medley system files. (The full filenames can also be used--the pseudohost just provides a systematic abbrevation.) If PSEUDOHOST is called with the host of a previous invocation but a different prefix, the new prefix replaces the old. If the prefix is NIL, the pseudohost is removed. If `HOST` is a list and `PREFIX` is `NIL`, `HOST` is interpreted as a (host prefix) pair.

The target host defaults to DSK if PREFIX does not have an explicit host.

When `PSEUDOHOSTS` is loaded, it executes

```
(PSEUDOHOST 'LI LOGINHOSTDIR)
```

so that files in the login directory can always be referenced succinctly with host `LI`, even while connected to another directory.

`(PSEUDOHOSTS)` [Function]

Returns the (host prefix) pairs of all currently defined pseudohosts.

`(PSEUDOHOSTP HOST)` [Function]

Returns the (host prefix) pair for a particular pseudohost, `NIL` if `HOST` is not a pseudohost.

`(TARGETHOST HOST)` [Function]

Returns the target host of a particular pseudohost, `NIL` if `HOST` is not a pseudohost.

`(TRUEFILENAME FILE)` [Function]

Returns the name of FILE in its true device, essentially replacing FILE's host by its prefix if it is a pseudohost. Returns FILE (possibly extended with the prefix of the connected directory) if its host is not a pseudohost. FILE may be a stream as well as a name.

(PSEUDOFILENAME FILE) [Function]

Returns the name of FILE in its pseudo device, if any, essentially replacing FILE's prefix by the hostname of a pseudodevice for that prefix. Returns FILE (possibly extended with the prefix of the connected directory) if it does not match a pseudohost prefix. FILE may be a stream as well as a name.

PS-SEND

By: Matt Heffron (mheffron@orion.cf.uci.edu)

Requires: POSTSCRIPTSTREAM, UNIXCOMM (only on Sun)

The module PS-SEND.LCOM is required by the PostScript ImageStream driver, and will be loaded automatically when POSTSCRIPTSTREAM.LCOM is loaded. It contains the function (POSTSCRIPT.SEND) which is called by SEND.FILE.TO.PRINTER to actually transmit the file to the printer. It is, by its nature, quite site specific, so it is in a separate file to make modifying it for any site relatively simple. For Sun Medley users, code has been added to send the completed Postscript files to the local printer (whatever the UNIX environment variable "PRINTER" is set to.)

POSTSCRIPT.SEND can handle the simple cases of copying a file to a spool directory or directly to a specific device (using COPYBYTES). The information about how to send a file to a specific host is expected to be on the SPOOLDIRECTORY, SPOOLFILE, SPOOLOPTIONS, HOST.CONTROL.STRING and HOST.CONTROL.AFTER.STRING properties of the host name. It checks first for the SPOOLFILE property on the host name, which must be a full filename that can be opened (by OPENSTREAM). If there is no SPOOLFILE property, then it checks for a SPOOLDIRECTORY property, if there, it will be concatenated together with a generated filename (by (GENSYM USERNAME)) and a ".PS" extension. If either the SPOOLFILE or SPOOLDIRECTORY properties exist, then an output stream will be opened onto the specified file. The value of the SPOOLOPTIONS property on the host name (if any) will be passed as the PARAMETERS argument to OPENSTREAM, and must be an appropriately formed list. (This is useful for cases where the specified destination is an 11xx device, such as {TTY} or {RS232} where you must set additional attributes of the stream like baud rate, etc.) If there is no SPOOLFILE or SPOOLDIRECTORY properties, then nothing will be sent and a message will appear int the PROMPTWINDOW ("[Unable to send FILE to HOST.]").

After the output stream is opened, if there is a HOST.CONTROL.STRING property of the hostname, then that string will be printed (IL:PRIN1) to the output stream first , then the first line of the file being sent (for a file generated by the PostScript ImageStream driver, this is the "%! ..." line), then the value of the POSTSCRIPT.CONTROL.STRING from the PRINTOPTIONS argument to POSTSCRIPT.SEND, finally the rest of the input file. (The idea of the HOST.CONTROL.STRING is that it should be a string to control the printing host itself, or perhaps a routing device that is mid-stream between the 11xx and the printer itself. For example, using a SPOOLFILE of "{TTY}FOO.PS" and having the PostScript printer shared by several additional computers (e.g. PC's) by use of a device like the Logical Connection from Fifth Generation Systems, it might be necessary to send a command to the Logical Connection to specify to route the output from this input to the output which is the PostScript printer.) Likewise, if there is a HOST.CONTROL.AFTER.STRING property of the hostname, then that string will be printed to the output stream last, just before closing the stream.

PS-SKETCH-PATCH

By: Will Snow (snow.envos@xerox.com)

Requires: SKETCH, POSTSCRIPTSTREAM

This module fixes some bugs in SKETCH when interacting with the PostScript driver. This will make the printing of sketches with text in them reasonable. It SYSLOADs SKETCH (from LISPUSERSDIRECTORIES) if it is not already loaded, it does not load POSTSCRIPTSTREAM.

PS-TTY

By: Matt Heffron (mheffron@orion.cf.uci.edu)

Requires: POSTSCRIPTSTREAM, PS-SEND, DLTTY

The module PS-TTY defines a printing host named PS-TTY which sends PostScript output to a printer over the {TTY} port of the 11xx. It also puts a function onto AROUNDEXITFNS which reinitializes the {TTY} after returning from LOGOUT. The BaudRate and other parameters of the {TTY} port are controlled by the following variables.

VARIABLES

PS-TTY-BAUD [InitVariable]

This is the BaudRate for the {TTY} port output stream. Defaults to: 4800.

PS-TTY-DATABITS [InitVariable]

This is the BitsPerSerialChar for the {TTY} port output stream. Defaults to: 8.

PS-TTY-PARITY [InitVariable]

This is the Parity for the {TTY} port output stream. Defaults to: NONE.

PS-TTY-STOPBITS [InitVariable]

This is the NoOfStopBits for the {TTY} port output stream. Defaults to: 1.

PS-TTY-FLOWCONTROL [InitVariable]

This is the FlowControl for the {TTY} port output stream. Defaults to: XOnXOff.

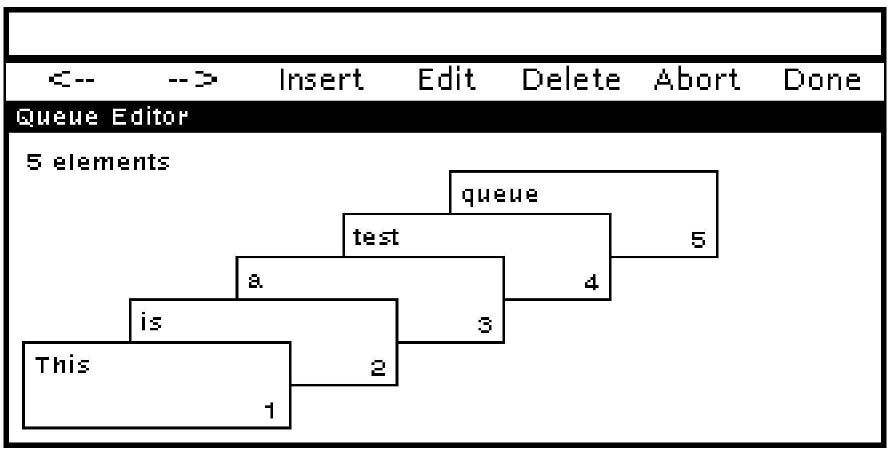
QEdit

By: Johannes A. G. M. Koomen
(Koomen.wbst@Xerox or Koomen@CS.Rochester.edu)

This document last edited on January 22, 1988

INTRODUCTION

QEdit is a facility for editing queues, which are ordered lists containing arbitrary objects. QEdit provides facilities for reordering, editing and deleting current queue entries, and inserting new entries. A QEditor looks like this:



Clicking the left mouse button over a queue entry makes it the current selection. You can then select a QEdit menu operation. QEdit returns with the original list if aborted, or a new list upon normal exit. Selecting an entry that is not entirely visible causes the display to scroll until it is.

DETAILS

(QEDIT *QUEUE PROPS*) [Function]

Invokes a QEditor on *QUEUE*. Returns *QUEUE* if aborted, a new list otherwise. The *PROPS* argument is a property list which serves to customize the behavior of QEDIT for this particular invocation. Currently defined props are listed below.

<— [QEdit command]

Moves the current selection forward, i.e., switches the current selection and the queue entry just before it.

—> [QEdit command]

Moves the current selection backward, i.e., switches the current selection and the queue entry just after it.

Insert [QEdit command]

Inserts a new entry in front of the current selection, provided *PROPS* contained an INSERTFN.

Edit [QEdit command]

Edits the current selection, provided *PROPS* contained an EDITFN.

Delete [QEdit command]

Deletes the current selection, provided *PROPS* contained a DELETEFN.

Abort [QEdit command]

Aborts the current QEdit session, returning the original queue.

Done [QEdit command]

Ends the current QEdit session, returning a new list containing the current queue entries.

TITLE [QEdit property]

If supplied, its value becomes the title for the QEdit window.

CONTEXT [QEdit property]

The value of the CONTEXT property is passed to the functions below as an extra argument. It does not affect QEdit directly. The functions below can also obtain the current queue by calling the function QEDIT.CURRENT.QUEUE (see below).

LABELFN [QEdit property]

If supplied, its value is a function which, when invoked on a queue entry and the user context, returns a label to use for displaying the queue entry. If not supplied, QEdit displays the queue entry itself.

LABELFONT [QEdit property]

If supplied, its value is a font specification for displaying the queue entry.

INSERTFN [QEdit property]

If supplied, its value is a function which, when invoked on the user context, returns either NIL or a new element to be inserted in front of the current selection (at the front of the queue, if there is no current selection). If not supplied, no elements can be inserted into the queue.

EDITFN [QEdit property]

If supplied, its value is a function which, when invoked on a queue entry and the user context, returns either NIL or a (possibly new) entry to be used instead of the current selection.

DELETEFN [QEdit property]

If supplied, its value is a function which, when invoked on a queue entry and the user context, returns NIL if the entry should not be deleted. If not supplied, no elements can be deleted. Hint: the function TRUE always returns T.

(QEDIT.CURRENT.QUEUE) [Function]

Invoked from one of the above mentioned functions, returns the queue being edited in its current form. The INSERTFN might use this, for example, if duplicates are not allowed.

(QEDIT.RESET) [Function]

QEdit will reuse QEditors upon reinvocation. This function will throw away any known but currently inactive QEditors. Useful if you wish to change the default QEdit props.

QEDITPROPS [QEdit variable]

Any props not explicitly supplied in the call to the function QEDIT are taken from this free variable. Its initial value is (TITLE "Queue Editor" LABELFONT (HELVETICA 8))

Examples

(QEDIT ’(This is a test queue)) [Function]

Brings up a QEditor as shown above. Queue elements can be rearranged, but not added to, edited or deleted.

(QEDIT ’(This is a test queue) ’(INSERTFN READ DELETEFN TRUE)) [Function]

Brings up a QEditor as shown above. Queue elements can be rearranged, inserted or deleted, but not edited.

READAIS

By: Nick Briggs (Briggs.pa@Xerox.com)

INTRODUCTION

AIS (array of intensity samples) is a format for color and gray- level images. The following functions allow reading and writing of AIS files from Lisp.

(AISBLT *FILE SOURCELEFT SOURCEBOTTOM DESTINATION DESTLEFT DESTBOTTOM WIDTH HEIGHT HOW FILTER NBITS LOBITADDRESS*) [Function]

Puts the image in an AIS file into a bitmap. AISBLT checks the sample size of the AIS file and the number of bits per pixel of the DESTINATION and performs the required reduction (if any). SOURCELEFT and SOURCEBOTTOM give the left and bottom coordinates in the source file of the image to be read (default to (0,0)). DESTINATION can be a bitmap, a color bitmap, or a window.

HOW indicates what method of reduction is to be used if the sample size of the AIS file is larger than the number of bits per pixel in the destination bitmap. The recognized methods are TRUNCATE (use the high-order bits) and FSA (use the Floyd-Steinberg dithering algorithm). The default when going to a 1 bpp bit map is FSA; otherwise it is TRUNCATE.

FILTER if non-NIL should be an array that will be used to filter the samples read from the AIS file. If FILTER is given and a sample point of intensity N is read from the file, (ELT FILTER N) is used to determine the bits for the destination. The function SMOOTHHIST described below is one way of getting a filter that balances the contrast in an image.

NBITS and LOBITADDRESS allow an image to be read into one or more ‘‘planes’’ of a color bitmap. NBITS tells how many bits are to be taken from each image sample, and LOBITADDRESS indicates the lowest bit within each pixel that the NBITS bits are to go. (Bit address zero is the leftmost or highest-order bit. For a four-bit-per-pixel bit map, three would be the lowest-order bit.) This is used by SHOWCOLORAIS to put the different planes of a color image into the bit map.

(SHOWCOLORAIS *BASEFILE COLORMAPINFO HOW SOURCELEFT SOURCEBOTTOM DESTINATION DESTLEFT DESTBOTTOM WIDTH HEIGHT*) [Function]

Reads a color image from three AIS files into a color bit map. The three color files are obtained by concatenating the strings ‘‘-RED.AIS’’, ‘‘-GREEN.AIS’’, and ‘‘-BLUE.AIS‘‘onto the end of BASEFILE. If COLORMAPINFO is a list of three small integers, it indicates how many of the bits in the destination are allocated to each color. For example, if DESTINATION is a four-bit-per-pixel color bit map and COLORMAPINFO is (1 2 1), one bit (bit zero) will be allocated to the red image, two bits (bits one and two) will be allocated to the green image, and one bit (bit three) will be allocated to the blue image.

DESTINATION is the color bitmap the image will be stored into.

HOW, SOURCELEFT, SOURCEBOTTOM, DESTLEFT, DESTBOTTOM, WIDTH, and HEIGHT are as described in AISBLT.

An experimental feature that is available only when going to 8 bpp color bit map: if COLORMAPINFO is a color map, each pixel will be determined by finding the color in the color map that is closest to the 24 bits of color information read from the three image files. (This takes a long time.) The function

COLOR.DISTANCE (red green blue redentry greenentry blueentry) is called to calculate the distance by which ‘‘closest’’ color is determined.

(CMYCOLORMAP *CYANBITS MAGENTABITS YELLOWBITS BITSPERPIXEL*) [Function]

Returns a color map that assumes the BITSPERPIXEL bits are to be treated as three separate color planes with CYANBITS bits being in the cyan plane, MAGENTABITS bits being in the magenta plane, and YELLOWBITS bits being in the yellow plane. Within each plane, the colors are uniformly distributed over the intensity range 0 to 255. White is 0 and black is 255.

(RGBCOLORMAP *REDBITS GREENBITS BLUEBITS BITSPERPIXEL*) [Function]

Returns a color map that assumes the BITSPERPIXEL bits are to be treated as three separate color planes with REDBITS bits being in the red plane, GREENBITS bits being in the green plane, and BLUEBITS bits being in the blue plane. Within each plane, the colors are uniformly distributed over the intensity range 0 to 255. White is 255 and black is 0.

(GRAYCOLORMAP *BITSPERPIXEL*) [Function]

Returns a color map containing shades of gray. White is 0 and black is 255.

(WRITEAIS BITMAP *FILE REGION*) [Function]

Writes the region REGION of the color bit map BITMAP onto the file FILE in AIS format. This provides an efficient way of saving color or gray- level images.

(AISHISTOGRAM *FILE REGION*) [Function]

Returns a histogram array of the region REGION in the AIS file FILE. The histogram array has as its Nth element the number of pixels in the region that have intensity N.

(GRAPHAISHISTOGRAM *HISTOGRAM W*) [Function]

Draws a graph of a histogram array in the window *W*. If *W* is NIL, a window is created.

(SMOOTHEDFILTER *HISTOGRAM*) [Function]

Returns a ‘‘filter’’ array that maximally distributes the intensities values contained in HISTOGRAM. The filter array can be passed to AISBLT to change the contrast of the image being read.

READAPPLEFONT

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: READDISPLAYFONT

READAPPLEFONT defines a new display font type which allows Envos Lisp to use (commercially available) Macintosh™ display fonts. Although this module is primarily intended for extracting font width information for conversion programs (eg. MacPaint™ to Sketch and vice versa) it can also be used to extend the number of display fonts available within the Envos Lisp environment.

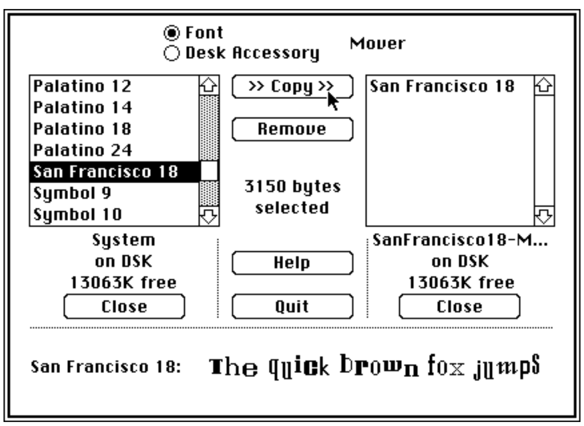
There are *no* user functions in the module. In addition to this module, you need a directory of extracted Macintosh™ font files (as described below) and you must add the name of that directory to the system DISPLAYFONTDIRECTORIES list. The following function is added under the type APPLE to the list DISPLAYFONTTYPES (defined by the READDISPLAYFONT module).

(READAPPLEFONT *STREAM FAMILY SIZE FACE*) [Function]

The module also adds the extension *APPLE* to the system list DISPLAYFONTEXTENSIONS.

FONT FILES

This module only uses the FontRec portion of the font files (see *Inside Macintosh*™). The font resources must be extracted into individual files with appropriate names, eg. SANFRANCISCO18-MRR-C0.APPLE. One method of doing this is to use the *Font/DA Mover*™ utility:



The individual font files should be moved to a Unix AUFS/CAP server (or the equivalent) and the resource forks (in the .resource directory) should be copied to the directory you added to DISPLAYFONTDIRECTORIES above.

If another method for extracting the FontRec data structure into individual files is used, the following variable will probably need to be reset:

APPLEFONTREC.OFFSET [Variable]

The offset in bytes into the font file of where the FontRec data structure begins (initially 264).

NOTES

- This module only handles proportionally spaced fonts and ignores fractional character widths.
- The user is responsible for determining the legality of extracting the fonts in question.

READBRUSH

By: Larry Masinter (Masinter.pa@Xerox.com)

uses: BITMAPFNS

This document last edited on September 8, 1988.

INTRODUCTION

This module implements two things:

(IDLE.GLIDING.BRUSH W box wait) [Function]

Like the default IDLE.BOUNCING.BOX Idle function but glides the bitmap around the screen instead of bouncing it.

(READBRUSHFILE file) [Function]

Reads files in the ".brush" format used by Mesa/Viewpoint. Returns a pair of bitmap/mask (or just (bitmap) if there is no mask. Brush file names use defaults from BRUSHDIRECTORY, initially {goofy:osbu north:xerox}<hacks>data>brushes> (and of use only inside Xerox.)

(READBRUSH file) [Function]

Calls READBRUSHFILE and then creates a window with that brush in it.

BRUSHDIRECTORY [Variable]

Default location to get brushes from.

Adds an entry to IDLE.FUNCTIONS for "Gliding box", which will use IDLE.GLIDING.BOX on the brush selected from a menu created by enumerating all of the .brush files on BRUSHDIRECTORY.

READDATATYPE

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

READATATYPE gives @ a read macro definition (in the **INTERLISP** readtable) so that it can be used to type in datatype pointers directly. For example, suppose you have lost your pointer to a window (or menu, etc.) but you have the printed representation around (eg. `{WINDOW}#56,17470`) then you can do things like:

```
90_(INVERTW @{WINDOW}#56,17470)
{WINDOW}#56,17470
```

The read macro is only intended to be used at the *read-eval-print* loop. If the character following the @ is not a **{** then the read macro returns the @ character just as if you had typed it in so that other expressions that use @, like `` `(A B ,@FILELST C D) ``, will still work correctly.

Although the read macro does not need the data type name in the brackets (eg. `{MENU}`) to get the pointer, it does require it in order to check the pointer to make sure it is of the correct *type*. If the pointer is not of the type specified, then the read macro returns NIL.

The following form is used in the COMS of the file to set the syntax of @ in the **INTERLISP** readtable and can be used to add the capability to other readtables and/or characters:

```
(SETSYNTAX '%@ '(MACRO FIRST READDATATYPE) (FIND-READTABLE "INTERLISP"))
```

READDISPLAYFONT

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

READISPLAYFONT modifies the display font functions to make it possible to define new display font types.

The functions \READDISPLAYFONTFILE and FONTFILEFORMAT are modified to use the list:

DISPLAYFONTTYPES [Variable]

An ALST containing font file extensions and the functions that can read those types from a file. Its initial value is:

```
((AC \READACFONTFILE)
 (STRIKE \READSTRIKEFONTFILE))
```

The functions take (STREAM FAMILY SIZE FACE) as arguments and return a CHARSETINFO datum. You will (probably) need the Xerox (internal) documentation about fonts and character sets (not supplied with the standard documentation) to define a new font file reading function.

The AC and STRIKE font types are handled specially to be compatible with the existing font code, so files with extension DISPLAYFONT still work and FONTFILEFORMAT moves the file pointer to the appropriately for those two types. For all other (new) types, the type is determined solely from the file extension and FONTFILEFORMAT has no side effects.

When defining a new display font types, you will need to add the new extension to the system list DISPLAYFONTEXTENSIONS.

REGION

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

REGION facilitates having multiple complex cursor behaviors in a single window without having the CURSORMOVEDFNs, CURSORINFNs, CURSOROUTFNs, and BUTTONEVENTFNs of the behaviors know about each other. In its simplest form it can be used to implement active regions.

To use, set the various window functions of the window to the REGION window functions and put a list of REGIONEVENT records on the REGIONEVENTLST property of the window. When the cursor moves over the window, REGION checks which region it is in, calls the CURSOROUTFN of the previous region and the CURSORINFN of the new region. If regions overlap, then the appropriate functions will be called on all regions affected by the mouse event.

The REGIONEVENTLST property of the window should contain a list of REGIONEVENT records which have the fields:

EVENTREGION	A REGION record which is the region of the window over which the region specific functions will be invoked.
REGIONBUTTONFN	(*WINDOW POSITION REGION REGIONEVENT*)
REGIONMOVEDFN	(*WINDOW POSITION REGION REGIONEVENT*)
REGIONINFN	(*WINDOW REGION REGIONEVENT*)
REGIONOUTFN	(*WINDOW REGION REGIONEVENT*)
REGIONREPAINTFN	(*WINDOW REGION REGIONEVENT*)
ACTIVEREGION	Boolean indicating if the region is active or not.
REGIONFLAGS	User defined identification flags.
REGIONUSERDATA	User defined field.

All of the fields in the REGIONEVENT record are optional. If a REGIONEVENT record has a NIL EVENTREGION, then it is considered the default REGIONEVENT and will be invoked whenever a mouse event occurs outside of any other region.

The REGION window functions are:

```
(WINDOWPROP WINDOW 'CURSORINFN (FUNCTION REGIONINFN))
(WINDOWPROP WINDOW 'CURSOROUTFN (FUNCTION REGIONOUTFN))
(WINDOWPROP WINDOW 'REPAINTFN (FUNCTION REGIONREPAINTFN))
(WINDOWPROP WINDOW 'CURSORMOVEDFN (FUNCTION REGIONMOVEDFN))
(WINDOWPROP WINDOW 'BUTTONEVENTFN (FUNCTION REGIONEVENTFN))
```

The above window properties can be set using the function:

(REGION.INIT *WINDOW [REGIONEVENTLST SAVE?]*) [Function]

The *REGIONEVENTLST* is a list of REGIONEVENT records to put on the window. If *SAVE?* is non-NIL, the CURSORINFN, CURSOROUTFN, etc. of the window are put into a default region event record (one with an EVENTREGION = NIL) and added to the *REGIONEVENTLST*. The macro:

(ADDREGIONEVENT *REGIONEVENT WINDOW*) [Macro]

can be used to add a *REGIONEVENT* record onto the current REGIONEVENTLST of *WINDOW*.

The REGIONFLAGS field of the REGIONEVENT record consists of whatever atoms the user wishes to identify regions with. These allow the user to issue commands such as “turn off all regions marked GRAPH”, “activate all the MENU regions”, etc.

(ACTIVATEREGIONS *FLAGS WINDOW*) [Function]

(DEACTIVATEREGIONS *FLAGS WINDOW*) [Function]

Activate and deactivate all the REGIONEVENT records on *WINDOW* whose REGIONFLAGS have a flag in common with *FLAGS*. If *FLAGS* is T, activate or deactivates all REGIONEVENT records.

DISABLEFLG [Variable]

If set to T, disables all of the region functions for all windows using the REGION module. Alternatively, setting DISABLEFLG to a window, or list of windows, disables all the windows using the REGION package except for those windows. This allows selectively turning off cursor actions on parts of the screen.

REGIONMANAGER

By:

Ron Kaplan

This document created in December 2021.

Medley comes equipped with a core set of functions for specifying regions and creating the windows that occupy those regions on the screen. But it can be disruptive if not irritating to have to draw out a new ghost region for every invocation of a particular application. Thus the common applications (e.g. TEDIT, SEDIT, DINFO...) implement particular strategies to reduce the number of times that a user has to sweep out a new region. They instead default to regions that were allocated for earlier invocations that are no longer active. TEDIT for example recycles the region of a session that was recently shut down, SEDIT allocates from a list of previous regions, DINFO always uses the same region, but FILEBROWSER always prompts for a new one. Applications that do recycle their regions tend to do so indiscrimately, without regard to the current arrangement of other windows on the screen or the role that those windows may play in higher-level applications.

The REGIONMANAGER package provides simple extensions to the core region and window functions. These are aimed at giving users and application implementors more flexible and systematic control over the specification and reuse of screen regions. It introduces three new notions:

A "typed region" allows the regions of particular applications to be specified, classified, and recycled according to their types.

The size, location, and orientation of a "relative region" is specified with respect to particular screen points and the location of other windows.

A "constellation region" encloses the collection of satellite windows (prompts, menus, etc) that surround the central window of an application.

REGIONMANAGER is innocuous in that explicit user action is required to change the default behavior of any system components.

Typed regions

REGIONMANAGER adds overlay veneers to the core `CREATEW`, `CLOSEW`, and `GETREGION` functions to make it easier to predict and control how different applications arrange their windows on the screen without always needing to respond to a ghost-region prompt.

The `REGION/INITREGION` arguments may now be region-type atoms in addition to either NIL or particular regions as `CREATEW` and `GETREGION` otherwise allow. The type-atom will resolve to a region drawn from a predefined pool of regions associated with that type, if the pool has at least one that is not currently allocated to another window. If the pool has no available regions, then the pool will be enlarged with a region that the user produces from a normal ghost-region prompt, and the type-atom will then resolve to the newly installed region.

A typed-region is marked as "inuse" and therefore unavailable when CREATEW assigns it to a window, and the extended CLOSEW marks it as again available when the window is closed.

An example of how an application can take advantage of this facility is the TEDIT-PF-SEE package. This provides lightweight alternatives to the PF and SEE commands that print their output to scrollable read-only Tedit windows, specifying PF-TEDIT and SEE-TEDIT as their region types. The user can predefine a preference-ordered sequence of recyclable regions that bring up multiple output windows in a predictable tiled arrangement, without region-prompting for each invocation.

The global variable TYPED-REGIONS is an alist that maintains the relationship between atomic type-names and the list of regions that belong to each type. The list is ordered according to preferences set by the user, and a type-atom is always resolved to the first unused region in its list. If the user is asked to sweep out a new region, that region is added at the end, as the least preferable. The function SET-TYPED-REGIONS is provided to add or replace TYPED-REGION entries.

(SET-TYPED-REGIONS TYPELISTS REPLACE) [Function]

TYPELISTS is an alist of the form

((type$_1$. regions$_1$)(type$_2$. regions$_2$)...)

where each region$_i$ is a possibly empty list of regions. For convenience, if TYPELISTS is just a literal type-atom, it is interpreted as ((type)), and if it is a list (type . regions) begining with an atom, it is interpreted as ((type . regions). The new regions replace preexisting regions if REPLACE, otherwise they are added at the front.

Typically, a call to SET-TYPED-REGIONS would be placed in a user's INIT file to set up the preference order for the regions that the user wants to participate in this reallocation scheme. If an application uses a type that is not on TYPED-REGIONS, then that type-atom is treated as NIL and always gives rise to the normal ghost-region prompting. Thus a user will observe no change in system behavior if TYPED-REGIONS is left with its initial value NIL. A type that is added with an empty region list (as opposed to not being on the list at all) will allow new regions to accumulate for recycling.

Relative regions

Two functions are provided to make it easy to create regions relative and oriented with respect to a specified reference point. These may be useful for constructing an application that includes a constellation of windows arranged in a particular relative way.

(RELCREATEREGION WIDTH HEIGHT CORNERX CORNERY REFX REFY ONSCREEN) [Function]

RELCREATEREGION creates a region of dimensions WIDTH and HEIGHT. One of its corners is identified by CORNERX and CORNERY and that corner will be aligned with a reference screen-point determined by REFX and REFY. If ONSCREEN, the WIDTH or HEIGHT will be adjusted with respect to that alignment so that the resulting region is entirely within the screen.

WIDTH and HEIGHT can be given as absolute (natural) numbers) or specified relative to the WIDTH and HEIGHT of another region or of the screen. The possibilities are interpreted as follows:

natural number: the number of screen points

list of the form (anchor fraction adjustment), where anchor is a region, window, or an atom SCREEN or TTY. The corres-ponding dimension of the anchor is mutiplied by fraction and adjustment is added to the result. For example, specifying (<window> .5 -1) results in a WIDTH that is one point smaller than half the width of window's region. Fraction and adjustment default to 1 and 0 respectively.

region/window/SCREEN/TTY: equivalent to (region/window/SCREEN/TTY 1 0).

CORNERX can be LEFT, RIGHT, or NIL=LEFT, CORNERY can be BOTTOM, TOP, or NIL=BOTTOM. If LEFT/TOP are specified, for example, the region will be splayed down and to the right of the reference point. If RIGHT/BOTTOM, then up and to the left.

The reference-point arguments REFX and REFY are interpreted as follows:

NIL: LASTMOUSEX/LASTMOUSEY

natural number: an absolute screen coordinate

(anchor fraction adjustment) or just region/window/SCREEN/TTY: the quantity determined relative to the size of anchor (as above) is added to the anchors left/bottom produce the REFX/REFY coordinate. In this case, fractions specified as LEFT/BOTTOM/NIL are interpreted as 0 and RIGHT/TOP are interpreted as 1. For example, a specification (<window> .4 -2) for REFY will produce a coordinate 2 points below the level that is 40% of the distance between the bottom and top of the window's region.

For convenience, if REFX is a position and REFY is NIL, then the XCOORD and YCOORD of REFX are taken as absolute values for REFX and REFY.

Also for convenience, if WIDTH is a potentially a list of RELCREATEREGION arguments, then the elements of that list are spread out in a recursive call.

(RELGETREGION WIDTH HEIGHT CORNERX CORNERY REFX REFY MINSIZE) [Function]

Calls GETREGION with an initial ghost region as created by RELCREATEREGION. CORNERX and CORNERY determine the ghost region's fixed corner, and the cursor starts at the region's diagonally opposite corner. If MINSIZE is true, then WIDTH and HEIGHT are taken as the minimum sizes of the region, except for adjustments that may be needed to ensure that all corners of the ghost region are initially visible on the screen.

(RELCREATEPOSITION REFX REFY) [Function]

Creates a position with X and Y coordinates specified by REFX and REFY references as above.

Constellation regions

Applications are often set up as a constellation of windows, a central or primary window surrounded by some number of satellites for menus, headers, prompts, and secondary outputs. The main panel of a file browser, for example, displays the list of files, but above it are carefully arranged windows for the column headers, summary information, and prompts, and off to the side is the menu of file browser commands. FILEBROWSER interprets the screen region that the user sweeps out for a new browser as the region for the whole constellation,the smallest region that will enclose the central window and all of its satellites. Similarly, the screen region given to TEDIT and SEDIT is divided between the prompt window and the central editing window, again so that the whole constellation (a pair in these cases) fit within the provided region.

Each of these applications is constructed by anticipating the subregions that the satellite windows will occupy after they are attached, decreasing the constellation region by their estimated (using WIDTHIFWINDOW HEIGHTIFWINDOW) or actual sizes, and then using remainder as the region for the central window.

An alternative approach is to construct the central window first, giving it the entire constellation region, and then to have ATTACHWINDOW reshape that window to accomodate the satellite windows as they are attached in sequence. This leads to the same final configuration, but there is no need for separate calculations to pre-adjust the region of the central window.

REGIONMANAGER provides an overlay veneer for ATTACHWINDOW that implements this strategy. If the new argument TAKEFROMCENTRAL is true, then the region of the WINDOWTOATTACH will be substracted from the region of the existing central window according to the EDGE parameter of the attachment.

(ATTACHWINDOW WINDOWTOATTACH MAINWINDOW EDGE POSITIONONEDGE WINDOWCOMACTION TAKEFROMCENTRAL) [Function]

This behavior is also triggered if the UNDERCONSTRUCTION property of the central window is true. Thus, a constellation can be set up by creating all of the satellites and the central window, marking the central window as under construction, and then doing the sequence of attachments. The property can be reset to NIL when the construction is complete, so the central window does not shrink if other other attachments (e.g. expanded menus) by later user actions.

A somewhat weaker form of a constellation is a collection of windows that are not attached around a central window but stand in a parent-child relationship at least with respect to closing and moving. A parent windows spawns children that respond independently to ordinary window commands (move, shape, close). But the children close when the parent closes, and the children move when the parent moves so that they continue to appear in the same relative positions. These primitives allow the construction of a tree of windows that are dependent in this way.

(CLOSEWITH CHILDREN PARENT) [Function]

Establishes a link between the PARENT window and any number of CHILDREN windows such that all CHILDREN will close when PARENT closes. The closing is accomplished by CLOSEWITH.DOIT:

(CLOSEWITH.DOIT PARENT) [Function]

Closes the close-with children of PARENT.

(MOVEWITH CHILDREN PARENT) [Function]

Establishes a link between the PARENT window and any number of CHILDREN windows such that all CHILDREN will move when PARENT closes. The closing is accomplished by MOVEWITH.DOIT:

(MOVEWITH.DOIT PARENT NEWPOS) [Function]

If NEWPOS is the new position of PARENT, moves each of the move-children so that they stand in the same relation to PARENT after it moves as before.

REMOTEPSW

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

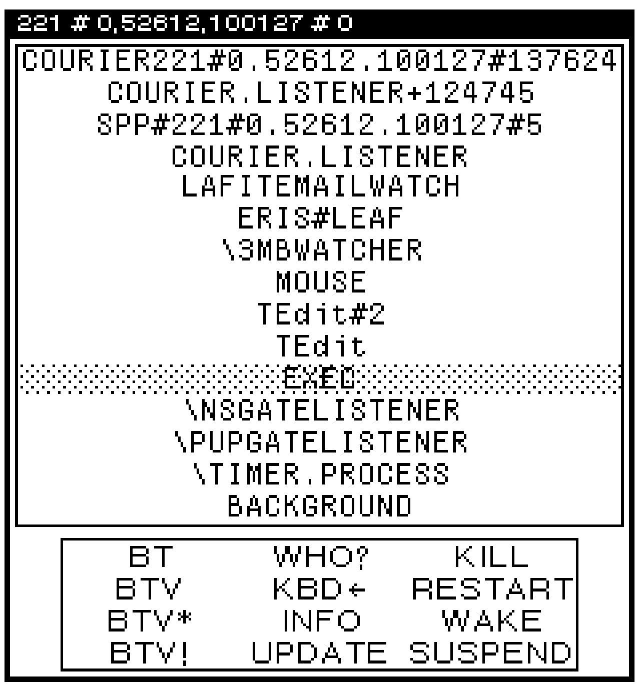
Uses: COURIERSERVE, COURIERDEFS

REMOTEPSW defines a *remote* process status window facility that runs on top of Courier. The remote process status window is identical to the local one except that it contains **UPDATE** (to get the current process status) instead of **BREAK**, and **INFO** is not implemented. Both the client and server code are contained in the module which must be loaded on both hosts. The Courier server must be running on the host you wish to monitor.

The only user function is:

(REMOTE.PROCESS.STATUS.WINDOW *HOST*) [Function]

which opens a remote process status window onto *HOST*, where *HOST* is any NS host specification that COURIER.OPEN accepts.



RPC

SUN REMOTE PROCEDURE CALLS

By: JFinger

Supported by Atty Mullins (Mullins.pa@Xerox.com) and Bill Van Melle (vanMelle.pa@Xerox.com).

This document last edited on August 1, 1988.

INTRODUCTION

This module implements SUN remote procedure calls as specified in the Remote Procedure Call Protocol Specification. The syntax is oriented toward Lisp users, differing greatly from Sun's C-like syntax.

RPC2 Package

All functions and variables mentioned in this document are defined as external variables in the package RPC2, unless otherwise stated.

REMOTE PROCEDURE DEFINITITION

Remote programs are defined via calls to define-remote-program.

define-remote-program	name number version protocol &key :constants [Function] :types :inherits :procedures Defines parameters and result types of the procedures of remote program (number, version, protocol) . If successful, returns name, otherwise nil.
name	a string or symbol that may be used by other procedures (for example, remote-procedure-call) to uniquely specify this remote program.
number	is the program number of this program on the remote machine. As specified in Sun's Remote Procedure Call Programming Guide, programs 0 - #x1fffffff are defined by Sun, #x20000000 - #x3fffffff are reserved for users, and #x40000000 - #x5ffffff are designated as transient.
version	a number, is the desired version of remote program.
protocol	an atom,UDP or TCP.(At the moment TCP is not supported under Medley 1.0-S).
constants	a list of pairs (<constant-name> <constant-def>), where <constant-name> is a symbol and a <constant-def> is an XDR constant (See XDR Constant Definitions below) .
inherits	a list of name 's of other remote programs from which types and constants are inherited. Inherited types and constants are resolved by searching this list in order.
types	a list of pairs (<type-name> <typedef>) , where a <type-name> is a symbol and a <typedef> is an XDR type definition (defined below).

procedures — a list of 4-tuples of the form (<procname> <procnumber> <arg-types> <result-types>), where <procname> is a symbol or string naming the procedure, <procnumber> is the procedure number on the remote machine, <arg-types> is a (possibly empty) list of XDR type definitions (see below) of the arguments to be sent to the remote procedure, and <result-types> is a (possibly empty) list of XDR type definitions of data to be returned from this remote procedure.

XDR (EXTERNAL DATA REPRESENTATION) TYPE DEFINITIONS

Because the client and server machines may represent data in different ways, a data representation common to both machines is necessary Remote procedure calls pass data between machines in 'External Data Representation' (XDR). The XDR language implemented here is oriented toward Lisp in its syntax and is not identical to the language spelled out in the Sun XDR Protocol Specification.

XDR data types may be defined in the :types keyword argument for later reference in the :types or :procedures of this or later remote programs. When a remote program is defined (usually at load time), the needed reading and writing functions are compiled for each constructed type referenced. Note that all XDR calls are eventually resolved to a composition of Primitive and Constructed XDR Type Definitions (see below).

SYNTAX

The keywords of the XDR language may be specified as symbols of the Keyword package.

All XDR Data Types Definitions (notated here as a <typedef>), used in Remote Procedure Calls are from the following language:

1) **Primitive Definition**:

One of the types in *xdr-primitive-types*,
:integer
:boolean
:unsigned
:hyperinteger
:hyperunsigned
:string
:float (not yet implemented)
:double (not yet implemented).

2) **Constructed Definition**:

One of the types in *xdr-constructed-types*,
(:enumeration (<symbol-1> <constant-1>) ...
(<symbol-n> <constant-n>))
(:union <enumeration-type> <typedef-1> ... <typedef-n>)
(:fixed-array <typedef> <constant>)
(:counted-array <typedef>)
(:opaque <constant>)
(:struct <defstruct-type> (<field-name-1> <typedef-1>) ...
(<field-name n> <typedef-n>))
(:sequence <typedef>)
(:list <typedef-1> ... <typedef-n>).

3) **Local Definition**:

A symbol defined previously in the same remote program definition.

Example: :types ((nrec :unsigned)...) says that type 'nrec' is really only of type ':unsigned'.

4) **Qualified Definition**:

A dotted pair of the form (<RPC program name> . <type>), where <type> is an XDR type local to <RPC program name>.

Example: :types ((count (myprog . nrec))...) says that a ’count’ is really whatever myprog defines a ’nrec’ to be.

5) **Inherited Definition**: A symbol defined in the :types argument of a remote program R such that R is on the list of remote programs passed as the :inherits argument to the current remote program definition. The first such type definition found is used, that is, the list of inherited programs is scanned from left to right.

XDR CONSTANT DEFINITIONS

Constants in XDR are defined by the following grammar:

<**constant-def**> ::= <**integer**> **|** <**defined-constant**>

<**defined constant**> ::=

<**locally defined constant**>
; Defined in the Remote Program currently being defined.

| <**inherited constant**>
; Defined in a remote program inherited by the current Remote Program (searched from left to right).

| <**qualified constant**>
; A dotted pair (<rp> . <constant>), where <constant> is defined in remote program <rp>.

SEMANTICS

An XDR type can be defined by a bidirectional filter mapping a subset of Lisp onto a byte stream and vice-versa.

For the XDR primitive type’s filter, a description is given of its argument on the Lisp and XDR sides.

:integer
Lisp: an integer in range -2,147,483,648 to 2,147,483,648 inclusive.
XDR: a 4 byte two’s complement integer, high order to low order.

:unsigned
Lisp: an integer in range 0 to 4,294,967,295 inclusive.
XDR: a 4 byte non-negative integer, high order to low order.

:boolean
Lisp: NIL for false, non-NIL for true. (The Lisp symbol T is returned when decoding a 1 from the XDR side.)
XDR: 0 for false, 1 for true.

:hyperinteger
Lisp: an integer in range -(263) to 263 -1 inclusive.
XDR: a 8 byte two’s complement integer, high order to low order.

:hyperunsigned
Lisp: an integer in range 0 to 264-1 inclusive.
XDR: a 8 byte non-negative integer, high order to low order.

:string
Lisp: a string of any length.
XDR: Suppose the string is of length n. The XDR representation is an :unsigned (the string length n) , followed by the n bytes of the string, followed by enough 0 bytes to make a multiple of 4 bytes.

:string-pointer
(UDP only)
Lisp: a dotted pair (addr . nbytes), where addr is a buffer’s address and nbytes is the number of bytes in the buffer. (Should I add an offset

argument?). This is a speed hack to avoid having to copy VMEMPAGEP's twice.
XDR: An XDR :string, as above.

:float — Lisp: A floating point number. (NOT YET IMPLEMENTED).
XDR: A 4 byte floating point number in IEEE format.

:double — Lisp: A floating point number. (NOT YET IMPLEMENTED).
XDR: A double precision floating point number in IEEE format.

:void — Lisp: null
XDR: no bytes.

For each constructed XDR type, the declaration syntax is given along with its corresponding mapping.

(:enumeration (<symbol> <integer>) ... (<symbol> <integer>))
Lisp: a symbol
XDR: an XDR :integer.
The Lisp symbol (Each symbol is the "discriminant" for that value of the enumeration) and the XDR integer will be from a corresponding pair in the declaration. It is an error to try to encode a symbol not in the declaration or to try to decode an XDR integer for which there is not a corresponding symbol in the declaration.

(:union <enumeration-type> (<symbol-1> <typedef-1>) ... (<symbol-n> <typedef-n>))
Lisp: A list of two elements, the first being a discriminant for the enumeration type, and the second the appropriate Lisp input/output for the typedef corresponding to that discriminant's type..
XDR: An :integer discriminant followed by the XDR input/output for the typedef corresponding to that discriminant's type.

(:fixed-array <typedef> <constant>)
Lisp: An array of length <constant>, each element of which is an object of type <typedefLisp>. Note that since the function elt is used in encoding, any Lisp sequence could be used in place of an array.
XDR: A sequence of <constant> objects of type <typedefXDR>.

(:counted-array <typedef>)
Lisp: A list of two elements, the first of which is an integer (the number of objects to be encoded/decoded), and the second of which is an array of objects of type <typedefLisp>.
XDR: An integer (the number of objects to be encoded/decoded) followed by that number of objects of type <typedefXDR>.

(:opaque <constant>) — Lisp: A string of length <constant>.
XDR: A sequence of <constant> bytes followed by enough null bytes to round <constant> up to a multiple of four.

(:struct <defstruct-type> (<field-name-1> <typedef-1>) ...(<field-name n> <typedef-n>))

Lisp: A struct of type <defstruct-type> such that each field mentioned in the this XDR declaration has a value. Note that a separate defstruct must be executed. The fields need not be named here in the same order as those in the defstruct, nor must all the fields named in the defstruct be used here.

XDR: A sequence of objects of types <typedef1 XDR>...<typedefn XDR>.

(:sequence <typedef>) This is fashioned after Courier 's method for encoding/decoding linked lists. This type can often be used to get around clumsy recursive definitions involving :union's of enumeration type :boolean.

Lisp: A list of objects of type <typedefLisp>.

XDR: A sequence of objects, each preceded by an XDR :boolean encoding of true. The last object in the sequence is followed by the XDR :boolean encoding of false.

Note: (:sequence <typedef>) produces the same encoding (but not the same decoding) as
(defstruct astructure this-element the-rest)
along with the declaration

```
(:recursive (:union     :boolean
                        (T (:struct astructure          (this-
                        element <typedef>)
                                                        (the-rest
                        astructure)))
                        (NIL :void))),
```

(:list <typedef-1> ... <typedef-n>)

Lisp: A list , the ith element of which is of type <typedefi Lisp>.

XDR: A sequence of objects, the ith of which is of type <typedefi XDR>.

(:skip <unsigned>) (For decoding only)

Lisp: Nothing

XDR: Any n bytes of data, where <unsigned> = n.

Note: This is a klooge for not having to decode the fattr's that NFS returns with every single cotton-pickin' memory read.

EXAMPLE OF A REMOTE PROGRAM DEFINITION

The following call to define-remote-program defines the portmapper remote procedures described in Sun's Remote Procedure Call Specification. Note that there are two definitions of procedure 4 given. Since remote procedures may be invoked by name, it is reasonable for there to be more than one definition for how to decode and encode the arguments to a given routine. In this case, both a recursive and non-recursive definition is given for the values returned from procedure 4. Note also that mapstruct and mapsequence must be defstruct'ed before this call to define-remote-procedure.

```
(define-remote-program 'portmapper 100000 2 'udp
        :types   '(  (mapstruct (:union  :boolean
                                (nil :void)
                                (t (:struct mapstruct
                                        (program :unsigned)
                                        (vers :unsigned)
                                        (prot :unsigned)
                                        (port :unsigned)
                                        (therest mapstruct)))))
                (mapsequence (:sequence (:struct mapsequence
                                                (program :unsigned)
```

```
                                                    (vers :unsigned)
                                                    (protocol :unsigned)
                                                    (port :unsigned)))))
      :procedures
            '(  (null 0 nil nil)
                (lookup 3 (:unsigned :unsigned :unsigned :unsigned)
                               (:unsigned))
                (gooddump 4 nil (mapsequence))
                (dump 4 nil (mapstruct))
                (indirect 5 (:unsigned :unsigned :unsigned
                               :string)
                               (:unsigned :string))))
```

UNDEFINING REMOTE PROGRAMS

undefine-remote-program name number version [Function]

MAKING REMOTE PROCEDURE CALLS

remote-procedure-call destination program procid arglist [Function]
&key destsocket version credentials protocol
dynamic-prognum dynamic-version
msec-until-timeout msec-between-tries noerrorflg

Performs a remote procedure call to program on destination. Returns a list of the returned values.

destination — Designates the host to which the procedure call is made. If Destination is a number it is interpreted to be the il:iphostadress of the host; if a symbol or string, it is a name from which the net address of the host may be found.

program — Designates the remote program to be called. If Program is a number, it is interpreted to be the remote program number. If a symbol, in which case it is assumed to be the name of the remote procedure (as defined in define-remote-procedure. If :version is non-nil, then program is treated as a number rather than as a name. If version is nil and program is a number, then the latest version of that program is used.

procid — Designates the procedure number from program to be called. If Procid is a number it is interpreted to be the remote procedure number; if a symbol, it is the name given that procedure in define-remote-procedure.

arglist — A list of the arguments to be serialized into XDR representation and passed as the arguments of the remote procedure call.

:destsocket — Normally, the remote socket must be looked up in the local caches (See *rpc-socket-cache* and *rpc-well-known-sockets*) or else found by making a call to the Portmapper on the remote machine. If :destsocket is non-nil, its value is used as the remote socket.

:version — If non-nil designated the desired version of program as well as causing program to be interpreted as a number rather than a name. See program above.

:credentials — An object of type authentication to be passed as the credentials of the remote procedure call. (See create-unix-authentication).

:protocol — A symbol specifying the transport protocol. Currently only UDP is implemented. Defaults to UDP. The only reason for using this parameter is to specify (along with the program and version), which known remote program is to be used.

:dynamic-prognum — If you really can't live without it, dynamic-prognum is used as the remote program number in spite of treating the arglist and returned values exactly as in program. Don't ask why.

:dynamic-version — If you really can't live without it, dynamic-version is used as the remote program version in spite of treating the arglist and returned values exactly as specified in program (and possibly version). Don't ask why. Defaults to 1.

:msec-until-timeout — Total number of milliseconds of waiting for a reply packet before giving up on this remote procedure call. Defaults to value of *rpc-msec-until-timeout*.

:msec-between-tries — Number of milliseconds between outgoing UDP packets. Defaults to *rpc-msec-between-tries*.

:errorflg — If :noerrors, ignores remote procedure call errors. If :returnerrors, returns the error as an s-expression. Otherwise, signals a Lisp error. Default t.

LOW-LEVEL REMOTE PROCEDURE CALL FUNCTIONS

setup-rpc — destination program procid &optional destsocket version protocol dynamic-prognum dynamic-version [Function]

Returns four values destaddr, socket, program and procedure (Yes, this is real, live multiple value return requiring a multiple-value-bind or something similar.) for consumption by perform-rpc. The arguments to setup-rpc are identical in meaning to the identically named arguments to remote-procedure-call.

open-rpc-stream — protocol destaddr destsocket [Function]

Returns an rpcstream for use by perform-rpc. Destaddr and destsocket are as returned by setup-rpc and protocol is identical to the protocol argument to remote-procedure-call.

close-rpc-stream — rpcstream protocol [Function]

Closes rpcstream, an rpc-stream of protocol protocol created by open-rpc-stream .

perform-rpc — destaddr destsocket program procedure rpcstream arglist credentials protocol &key errorflg leave-stream-open msec-until-timeout msec-between-tries [Function]

Performs a remote procedure call returning a list of the values retruned by the remote procedure.

LISTING REMOTE PROGRAMS CURRENTLY DEFINED

list-remote-programs [Function]

Returns a list of 4-tuples (name number version protocol) for each remote program currently defined.

CREATION OF CREDENTIALS

create-unix-authentication
stamp machine-name uid gid gids [Function]

Returns a Unix-type authentication suitable for use as the credentials of a call to remote-procedure-call or perform-rpc.

stamp	An arbitrary unsigned integer.
machine-name	A string containing the name of the calling machine.
uid	User id number on the remote machine.
gid	Group id number on the machine.
gids	A list or array of group id numbers (on the remote machine) that contain the caller as a member.

GLOBAL VARIABLES

xdr-primitive-types	An a-list of keywords and the corresponding function that implements that XDR primitive type.
xdr-constructed-types	An a-list of keywords and the corresponding function that generates code to implement that XDR constructed type.
msec-until-timeout	Number of milliseconds before giving up on receiving a reply packet. Default 1000.
msec-between-tries	Number of milliseconds to wait before resending UDP packet. Default 100.
rpc-ok-to-cache	If non-nil, uses *rpc-socket-cache* as a cache of socket numbers found to date.
rpc-well-known-sockets	A list of well-known sockets. Format is (<host address> <remote program number> <remote program version> <protocol> <socket>)
rpc-socket-cache	A list of non-well-known sockets. Format is same as *rpc-well-known-sockets*.
debug	If non-nil prints out debugging information. If a number, the higher the number, the more information is printed. Default nil.

RPC FILES

RPC	Sets up the RPC2 Package and loads other RPC files. Loads Portmapper remote program definition and executes it.
RPCLOWLEVEL	Super low-level UDP/TCP functions added to Eric Schoen's TCPUDP code.
RPCOS	Low-level interface to Sun OS networking code .
RPCSTRUCT	Structure definitions used by the other files. These are in a separate file because they take so long to compile.
RPCCOMMON	Common lookup functions and stream i/o functions used by the other files.
RPCXDR	External Data Representation (XDR). Code Generation for XDR constructed types and XDR primitive functions.
RPCRPC	Remote program definition and remote procedure calls.
RPCPORTMAPPER	Definition of portmapper in UDP and TCP.

KNOWN DEFICIENCIES

Floating point XDR types are not implemented.

The view-packet utility is not documented and needs to be smarter about authentications.

Fall through cases of XDR types UNION and ENUMERATE should be added.

TCP is not supported under Medley 1.0-S, this should be in the next release.

COPYRIGHT INFORMATION

Written by Jeff Finger under support of National Institutes of Health Grant NIH 5P41 RR00785

to the SUMEX-AIM Computing Resource at Stanford University.

Modified to work under Medley 1.0-S by Atty Mullins.

RS232CNetwork

By: Nick Briggs (Briggs.pa@xerox.com)

Uses: DLRS232C

This document last edited on August 10, 1988.

INTRODUCTION

The RS232C port on a Daybreak can be configured in such a way that it can be used to communicate with a Pup Gateway over a phone line to provide a network communications path equivalent in all but speed to an Ethernet connection.

USE

Load the RS232CNetwork module. It redefines a number of functions, the one of general interest being

(RS232C.INIT *BAUDRATE BITSPERSERIALCHAR PARITY NOOFSTOPBITS FLOWCONTROL LINETYPE*) [Function]

RS232C.INIT performs as documented in the Lisp Library modules manual, except for the addition of the *LINETYPE* argument. *LINETYPE* should be SYNC or ASYNC (also accepted are the alternate spellings SYNCH and ASYNCH).

In order to use the RS232C port for communicating with a Pup Gateway it must be configured (using RS232C.INIT) in SYNC mode. In addition, the variable

RS232C-NETWORK [Variable]

should be set to T (which is the default setting on loading the RS232CNetwork module.

The variable

RS232C-NETWORK-AUTODIAL [Variable]

controls whether the system will attempt to start the network connection automatically when Lisp returns from a LOGOUT. If *RS232C-NETWORK-AUTODIAL* is set to NIL, you can cause the system to attempt to establish the connection manually by calling the function RESTART.ETHER.

The modem should be configured to dial on detecting an off to on transition of DTR. On a Codex 2260 modem this is DTR mode 108.1. Since the dialing and answering process can take a non-trivial amount of time, the variable

RS232C-NETWORK-DIALING-TIMEOUT [Variable]

indicates how many seconds the code will wait for the modem to signal that the connection has been established (by raising DSR) before timing out and continuing without the network connection. The default value is 30 seconds.

EXAMPLE

The very first time the code has been loaded the initialization sequence, when using Codex 2260 modems which can communicate at 9600 baud, would be

```
(RS232C.INIT 9600 8 'NONE 1 NIL 'SYNC)

(SETQ *RS232C-NETWORK-AUTODIAL* T)

(RESTART.ETHER)
```

Subsequently, whenever the Daybreak boots into Lisp, the network code will attempt to establish a phone connection.

To shutdown the phone connection you can call the function TURN.OFF.ETHER. The connection will also be broken when you log out of Lisp (assuming that the modem is configured to disconnect on loss of the DTR signal, which the Codex 2260 modems will do)

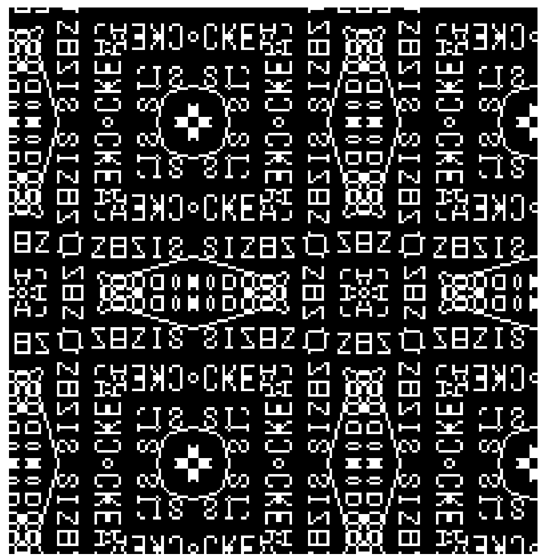
SCREENPAPER

By: Larry Masinter (Masinter.pa@Xerox.com)

SCREENPAPER is an Idle hack ("Screen wallpaper"). Old fashioned wallpaper/wrapping paper from your screen.

Global parameters:

SCREENPAPERSIZE	size of viewport, initially 64.
SCREENPERIOD	how often to go into reflective move
SCREENREPEAT	how long to stay in reflective mode (initially 0. e.g., disable reflective mode).



SEARCHMENU

By: John Maxwell (Maxwell.pa)

Uses: DICTTOOL, DICTCLIENT, ANALYZER

INTERNAL

This document last edited on October 16, 1987

INTRODUCTION

The SearchMenu package implements a user interface to the relevance search capabilities of the dictionary server. Relevance search is a technique for finding items by giving examples of what you are looking for and having the search algorithm look for "similar" items. These items are then displayed in a FreeMenu which allows you to select those items which are actually relevant. You can then iterate on the new set of examples until no new relevant items show up.

The Dictionary Server currently supports three databases that can be searched using this technique: the WordNerd (based on the American Heritage Dictionary), the EtymologyNerd (based on the etymological portion of the American Heritage Dictionary), and the IRMNerd (based on the Interlisp Reference Manual). For example, if you were looking for for different types of gases, setting the database to the WordNerd and giving it example gases such as hydrogen and helium would produce oxygen, xenon, krypton, argon, neon, radon, flourine, chlorine, and nitrogen. Or, if you were looking for Interlisp functions that tested the equality of something, setting the database to the IRMNerd and giving example procedures like EQ and EQUAL would produce EQP, IEQP, FEQP, STREQUAL, EQMEMB and EQUALALL.

HOW TO USE THE SEARCH MENU

When you load the SearchMenu package, a Search Menu will appear in the lower left corner of the screen. (The Search Menu expands to display the results of a search, so you might leave it there until you see how big it can get.) The Search Menu has a number of commands on the top line, followed by a place to type in examples.

To try your first search, click the "Examples:" field and type "lion tiger". Then click the MATCH WORDS! command just above it. A little menu of databases will appear, asking you which database you want to search in. Select "DictServer: WordNerd". A message will then appear in the prompt window saying: "Searching in DictServer: WordNerd for words like: (tiger lion)." After about 30 seconds a new Search Menu will be created with a list of items that the Dictionary Server thinks is similar to a lion and a tiger. At the top of the list lion and tiger will already be selected.

To continue the search, scan the list of items for things that are relevant to you. If you are not sure what a word means, click the "def" button to its left, and the definition will be printed out in a separate window. If you want an item to be included in the next search, simply click at it and it will become inverted, like lion and tiger. If you click at an item twice, a line will be drawn through it to indicate that any keywords that it uses should have their weights reduced. Words that are neither highlighted nor struck out are ignored. When you are through examining the items, invoke the MATCH WORDS! command. You can iterate like this as often as you like.

Clearing The Search Menu

Before you start a new search, you should invoke the "CLEAR!" command to clear the Search Menu. The Search Menu caches some information about the search you are conducting which may interfere with your next search. Therefore, to be sure that you get a clean search, you should clear the menu.

Changing Databases

When you want to search for items in a new database, all that you need to do is click the "SET DATABASE!" command. This will cause a menu of databases to appear. Clicking one of the items in the menu will cause the Search Menu to search in that database from then on. Clicking outside the menu leaves the Search Menu in its current state.

The Key Menu

To the left of the commands in the first line is a command labelled "KEY MENU!". Invoking this command will turn the Search Menu into a key menu. The key menu will have all of the keys that were used in the last search, along with their weights. The weights are editable. If you want to see what would happen if you searched using different weights, simply edit the weights and then invoke the "MATCH KEYS!" command. Weights that are set to 0 or have no value are ignored.

If you want to see all of the uses of a particular key click the "uses" button to the left of that key. To get back to the example menu, invoke the "SAMPLE MENU!" command in the upper left of the Search Menu.

Logging a Search

If you wish to keep a log of each search, set the SearchMenu.LogData variable to T. From then on, the Search Menu will write the results of its search into a private stream. When you click the "CLEAR!" command, a TEdit window will be opened on the log and the log cleared in preparation for the next search.

THE PROGRAMMER’S INTERFACE

The Dictionary Server can be accessed directly through the following procedures:

(DICTCLIENT.MATCHWORDS *POSWORDS NEGWORDS MINWORD MAXWORD DICTIONARY*) [Function]

Takes a list of words to match, plus a list of words to ignore, plus the range of words that you are interested in, plus the database. The range of words lets you look at a search one chunk at a time: the first fifty (MINWORD = 1, MAXWORD = 50), then the next fifty (MINWORD = 51, MAXWORD = 100), and so on. The DICTIONARY is the name of the database to be searched in. Currently there are three possibilities: 'WordNerd, 'EtymologyNerd, and 'IRMNerd. If no data base is specified, the default is 'WordNerd.

(DICTCLIENT.WEIGHTEDSEARCH *WEIGHTEDKEYS MINWORD MAXWORD DICTIONARY*) [Function]

Takes a list of weighted keys to match, plus the range of words that you are interested in, plus the database. The weighted keys should be of the form '((key1 weight1)(key2 weight2)...). The range of words lets you look at a search one chunk at a time: the first fifty (MINWORD = 1, MAXWORD = 50), then the next fifty (MINWORD = 51, MAXWORD = 100), and so on. The DICTIONARY is the name of the database to be searched in. Currently there are three possibilities: 'WordNerd, 'EtymologyNerd, and 'IRMNerd. If no data base is specified, the default is 'WordNerd.

SEdit-Menu-Always

By: Matt Heffron (BEC.HEFFRON@ECLA.USC.EDU)

This package advises the SEdit Editor so that when an SEdit window is opened, the SEdit Attached Command Menu is automatically opened as well (depending on the setting of the global variable IL:SEditMenuAlwaysFlg). The value of IL:SEditMenuAlwaysFlg is initialized to T as an INITVAR when the file is loaded.

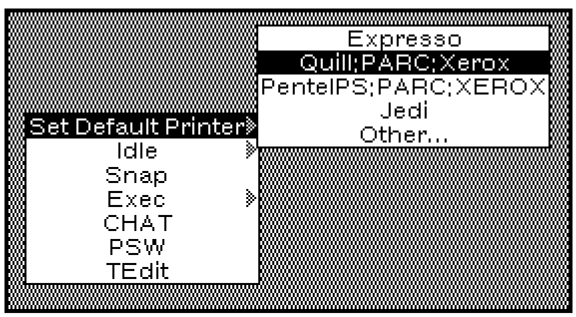
SETDEFAULTPRINTER

By: Nick Briggs (Briggs.pa@Xerox.com)

The SETDEFAULTPRINTER module provides a (cleaner) mechanism for moving printer names around on your DEFAULTPRINTINGHOST list. There are no user callable functions. Access to the features of the module are through the Background menu. This module uses the DEFAULTSUBITEMFN module which redefines the DEFAULTSUBITEMFN used in menus to accept an expanded form for menu subitems.

Set Default Printer [Background Menu Entry]

Selecting the "Set Default Printer" item off the background menu will prompt you for a new default printer, which will be added at the beginning of the DEFAULTPRINTINGHOST list. If you roll-out into the subitems for Set Default Printer it will present a submenu with the entries on DEFAULTPRINTINGHOST, and an "Other..." item. Selecting one of the printer name entries will cause it to be moved to the front of DEFAULTPRINTINGHOST, selecting "Other..." will prompt for the name of a printer in the same manner as selecting the "Set Default Printer" top level item off the background menu. If any commentary information has been supplied (see below) holding the mouse over the printer name will display the information in the prompt window.



SDP.PRINTERINFO [Variable]

The variable SDP.PRINTERINFO is an A-list which will be used to lookup commentary information about a printer to be included as the "help" in the menu subitems. The UPPERCASE name of the printer is used as a key. An example SDP.PRINTERINFO setting might be

```
((QUAKE . "Press, Rm 1532") (PENTELPS:PARC:XEROX . "Interpress, Rm 1532"))
```

LOCATION [Property]

The code that looks up the commentary information about a printer will also check for a LOCATION property on the UPPERCASE atom which is the printername if no entry is found on SDP.PRINTERINFO. For example

```
(PUTPROP 'JEDI 'LOCATION "FullPress, Pod 5, 2nd floor")
```

Would describe the location of printer Jedi.

SHOWTIME

By: Timothy Bigham (TBigham.henr@Xerox.com)
Medley mods by: Ron Fischer (Fischer.PA@Xerox.com)

Uses: BITMAPFNS, SCALEBITMAP, READBRUSH

This document last edited on May 13, 1988.

INTRODUCTION

SHOWTIME provides a user interface to read, write, and edit bitmaps in several different formats. Among the supported formats is RES, used in VIEWPOINT Freehand Graphics. Other supported formats include Brush (Mesa Doodle format); and Lisp.

SHOWTIME has been written to readily accomodate new formats. Users may add new formats to Showtime by writing format-specific read and/or write functions and adding them to those Showtime knows about (described below).

Selecting SHOWTIME from the background will provide the user the opportunity to specifiy and area to use as the SHOWTIME window. After the user creates a SHOWTIME window, a left mouse button within the window will popup a menu of available options. There may only be one bitmap displayed in a SHOWTIME window at a time, but any number of SHOWTIME windows may be opened. Shrinking a SHOWTIME window will create an icon with the name of the bitmap that is displayed in the window.

Functions, Variables, and Lisp Code Examples

SHOWTIME.FORMAT.FNS [Variable]

A global association list that maintains a list of all the formats Showtime knows about and the read and write functions to use with those formats. This variable should be updated by calling the function SHOWTIME.ADD.FORMAT to ensure successful integration of any new bitmap formats.

SHOWTIME.DEFAULT.FORMAT [Variable]

A variable that is initially set to 'LISP. This format uses the binary storage routines found in the lispusers module BITMAPFNS.

(SHOWTIME.ADD.FORMAT *FORMAT READFN SAVEFN*) [Function]

A function that should be called when the user wants Showtime to know about new bitmap formats. *FORMAT* may be any descriptive atom, such as RES or LISP. *READFN* and *SAVEFN* must be functions that have as the first two arguments FILENAME and BITMAP. In addition, the READFN must return a bitmap.

For example, the READFN code for LISP format is:

```
(LAMBDA (FILENAME)

(* this function must <1> read a bitmap from a file and <2> return the value
of the bitmap)

      (READBM (OPENFILE FILENAME (QUOTE INPUT))))
```

For example, the WRITEN code for LISP format is:

```
(LAMBDA (FILENAME BITMAP) (* TBigham "30-Dec-86 13:09")

(* this function must write a bitmap to a file)

      (WRITEBM (OPENFILE FILENAME (QUOTE OUTPUT)) BITMAP))
```

ACKNOWLEDGEMENTS

SHOWTIME was originally developed to provide a user interface in the exchange of bitmaps between the VIEWPOINT and INTERLISP-D environments. Tom Wall initiated the idea to exchange bitmaps between these environemts and was instrumental in developing the code to write RES files. Gary Gocek originally wrote the code to read RES files; Mitch Garnaat wrote the code to write BRUSH files; reading brush files is supported by the lispusers module READBRUSH written by Larry Masinter. The SHOWTIME icon was designed and created by Mary Baecher-Cocca.

SIMPLECHAT

By: Larry Masinter (Masinter.PA@Xerox.COM)

Uses: `TEDIT`

This document last edited on Sept. 8, 1988.

INTRODUCTION

Like CHAT except that it works in the current window/exec instead of spawning a new window. To exit from TTYCHAT there is an escape character, control-right-bracket (^]). If you type ^], you get prompted for a Chat command. This can be one of Binary, Text, or Close. Normally TTYCHAT translates incoming characters and converts EOL; setting Binary mode disables this. Close will close the connection.

MODULE EXPLANATIONS

The CHATSERVER module advises CHAT to use TTYCHAT when the main "terminal" is not the display. This allows one to use the Lisp system as a "protocol translation gateway"; for example, on a Sun with CHATSERVER-NS loaded, you can Chat to the Sun using NS and then use UNIXCHAT to CHAT(SHELL).

(TTYCHAT *&optional host logoption*) [Function]

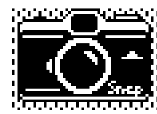
SNAPW-ICON

By: Randy Gobbel (Gobbel.pa)

Uses: nothing but basic window functionality

This document last edited on September 8, 1988.

INTRODUCTION

SNAPW-ICON creates an icon for shrunken screen snap windows. It looks like this: 
Everything else about screen snaps is as before.

SOLID-MOVEW

By: Lennart Lövstrand
(Lovstrand.EuroPARC@Xerox.COM)

This document last edited on May 13, 1988.

INTRODUCTION

This module changes the behaviour of MOVEW when no destination is given to let the whole image of a window track the mouse instead of just its outline. To avoid flickering and to give an illusion of a smooth animation, all rendering operations are done off the screen, ending with a single bitblt to the frame buffer for each cycle. This can easily be done on small windows such as icons, but the more bits there are to be moved, the longer it takes to do the animation updates and the slower it becomes to solidly move windows. Therefore, the user can control when solid vs. outline moving is to be done by setting *SOLID-MOVEW-FLAG* to an appropriate value. By default, only windows containing less than 15,000 pixels will be moved solidly; all other windows are moved using the original MOVEW method.

SOLID-MOVEW interfaces nicely with both ICONW and ATTACHEDWINDOWS by being able to move images of arbitrary shape — not just pure rectangles. It also knows about GRID-ICONS and can be made to force the icons to snap to grid positions while being moved, thus producing a kind of jagged feeling. Finally, a shadow has been added emphasize the 2½-D property of window systems and to give a clear indication of when the window is in the process of being moved.

PROGRAMMER'S INTERFACE

When loaded, the module replaces the system MOVEW function with its own version and moves the original code to ORIGINAL-MOVEW. The control and interaction is then comes through the following variables:

SOLID-MOVEW-FLAG [Variable]

This variable controls whether the new MOVEW should use solid or outline moving. It should have one of the following types of values:

a NUMBERP	Only use solid moving on windows that have a total size (width x height) less than or equal to the given number of pixels.
a POSITIONP	Only use solid moving on windows that have a width and height less than or equal to the two numbers.
ICON	Only move icons solidly. A window is considered to be an icon if it either has an ICONFOR or an ICONIMAGE property.
T	Move all windows solidly.
NIL	Move all windows using outlines.

The default value for *SOLID-MOVEW-FLAG* is 15000.

SOLID-MOVEW-SHADOW [Variable]
SOLID-MOVEW-SHADOW-SHADE [Variable]

These two variables define whether or not a shadow should accompany the moving image. The shadow is always directed towards south-east and the first variable, *SOLID-MOVEW-SHADOW*, determines its position by taking on any of the following types of values:

a NUMBERP	The x and y offsets of the shadow (same)
a POSITIONP	The x and y offsets of the shadow (different)
T	Use the default shadow offset — 3 pixels in both directions.
NIL	Don’t show a shadow.

The second variable, *SOLID-MOVEW-SHADOW-SHADE*, sets the darkness of the shadow, ie. the texture to be added to the background where the shadow is visible.

The default values for the two variables are T and 42405, a 50% gray shadow offset by 3 pixels.

SOLID-MOVEW-GRIDDING [Variable]

When used together with the ICON-GRIDS module, SOLID-MOVEW can be made to only move solid window images on grid positions, thus creating a kind of "jagged" feeling when interactively moving icons on the screen. If this is disabled, the icon will "snap" to the closest grid position only after the move has been completed.

The default value for *SOLID-MOVEW-GRIDDING* is NIL, thus disabling early gridding.

SOLID-MOVEW-CASHING [Variable]

SOLID-MOVEW uses separate bitmaps for rendering purposes so as to produce a smooth animated move and avoid unnecessary flickering on the screen. To speed up the initial phase of the move operation, the rendering bitmaps can be cached from one invocation to another. This will use up some bitmap space, but can be freed using (GAINSPACE) if need arises.

The default value for *SOLID-MOVEW-CASHING* is T, thus enabling cached rendering bitmaps.

(SOLID-MOVEW *POSorX Y*) [Function]

Because only those windows meeting the requirements of *SOLID-MOVEW-FLAG* will be moved solidly, the user has the option of calling SOLID-MOVEW. It takes the same arguments as MOVEW, but if either *POSorX* or *Y* is specified, control is again turned over to the old MOVEW.

If you get tired of all this, you can undo the behaviour of SOLID-MOVEW by typing the following form into an Interlisp Exec:

```
(MOVD 'ORIGINAL-MOVEW 'MOVEW)
```

BUGS

No provision has been made to make SOLID-MOVEW work with color.
If the window is closed as a side effect of the its MOVEFN or AFTERMOVEFN, it will be reopened before SOLID-MOVEW returns

SOLITAIRE

By: Beau Sheil. Upgraded for Medley by Larry Masinter (Masinter.PA@Xerox.COM)

The SOLITAIRE package ia a simple graphics demonstration program that plays and animates the solitaire card game (known as ~Patience~ in English speaking countries). Solitaire is a game for one, so there is no way to play ~against~ the machine. SOLITAIRE is most effective as a background activity when the machine is doing nothing else, so it is frequently used as an IDLE hack.

TO USE

To play once

(SOLITAIRE *SOLOW REPLAY*) [Function]

Plays one hand of solitaire, which it will animate in the window *SOLOW* (which should be at least 700 by 700, although the program will do its best to adapt). If *REPLAY* is T, SOLITAIRE will use the deck from the previous shuffle, else it will deal a new hand.

To play repeatedly

(SOLO *SOLOW*) [Function]

Calls (SOLITAIRE *SOLOW*) repeatedly.

The results

SOLO keeps a record of the frequency of each of its results in the array SOLORESULTS [0..52] which it plots at the end of each hand.

As an IDLE hack

Loading SOLITAIRE automatically adds SOLITAIRE as an option to the IDLE menu. If chosen, it will be given the ~whole screen~ covering window of IDLE and will use a black background, rather than its usual shaded one, to preserve the screen phosphor. Otherwise, its operation is completely normal.

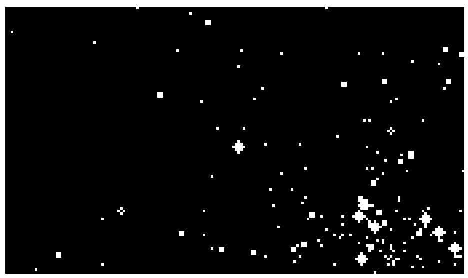
STARBG

By: Gregg Foster (Foster.PA@Xerox.COM)

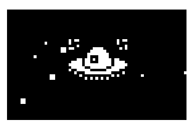
Upgraded for Medley by Larry Masinter (Masinter.PA@Xerox.COM)

STARBG creates a random star field for your screen background and and a little flying saucer to follow your cursor when it's in space (so it doesn't get lost). It also supplies an alternate IDLE function, Cosmos.

The star field will look something like this:



The saucer will look like this:



USAGE

(STARBG) [Function]

STARBG fills a screensized bitmap with random stars, turns the saucer on, and calls CHANGEBACKGROUND. If you don't like the star pattern you get, try it again.

(Cosmos window) [Function]

Cosmos is puts an evolving universe in a window. It's intended as an IDLE function, but will entertain you for hours in any decently sized window.

(SaucerOn) [Function]

SaucerOn turns the saucer on by changing the CURSORBACKGROUND*FNs.

(SaucerOff) [Function]

SaucerOff turns the saucer off and sets the BACKGROUNDCURSOR*FNs to NIL.

CUSTOMIZATION

There are lots of user-settable parameters, all of which have reasonable defaults. Here are some of the interesting ones:

STARBGParameters [Variable]

is a list of settable parameters. Most are dotted pairs specifying ranges (e.g. stars3 defaults to (6 . 70) meaning that STARBG will make 6 to 70 type-3 stars). The others are bitmaps.

BM1, ..., BM5 [Variables]

The star bitmaps used to BLT the stars. BM1 must be a single bit.

SBM [Variable]

The starry screen bitmap. This is reused in subsequent calls to STARBG.

stars1, ..., stars5 [Variables]

Ranges for the 5 kinds of stars.

constellations [Variable]

Range for number of constellations. A constellation is a group of bright stars.

clusters [Variable]

Range for number of clusters. Clusters are tightly globular.

superClusters [Variable]

Range for number of superClusters. SuperClusters are clusters of clusters.

eventPause [Variable]

Number of milliseconds to block between events. Larger numbers have the effect of slowing down the rate of evolution..

changeStars [Variable]

Will use the IDLE-ing star field as your new background.

STEP-COMMAND-MENU

By: Matt Heffron (BEC.HEFFRON@ECLA.USC.EDU)

This package changes the function CL::STEP-COMMAND (used by CL:STEP) to call a new function (instead of IL:ASKUSER) to get its commands from a menu attached to the stepping window (depending on the setting of the CL:SPECIAL variable IL:*STEP-COMMAND-MENU*). The value of IL:*STEP-COMMAND-MENU* is initialized to T as an INITVAR when the file is loaded. The variable USER::*STEP-COMMAND-INVERT-MENU-SHADE* is the shade used to *grey-out* the attached menu when the stepping is not awaiting a command. The menu is attached to the Right edge (at the Bottom) of the stepping window. (If there isn't enough room on the Right, it will be attached to the Left edge.) The menu is detached and closed when the stepping level which first attached it is exited.

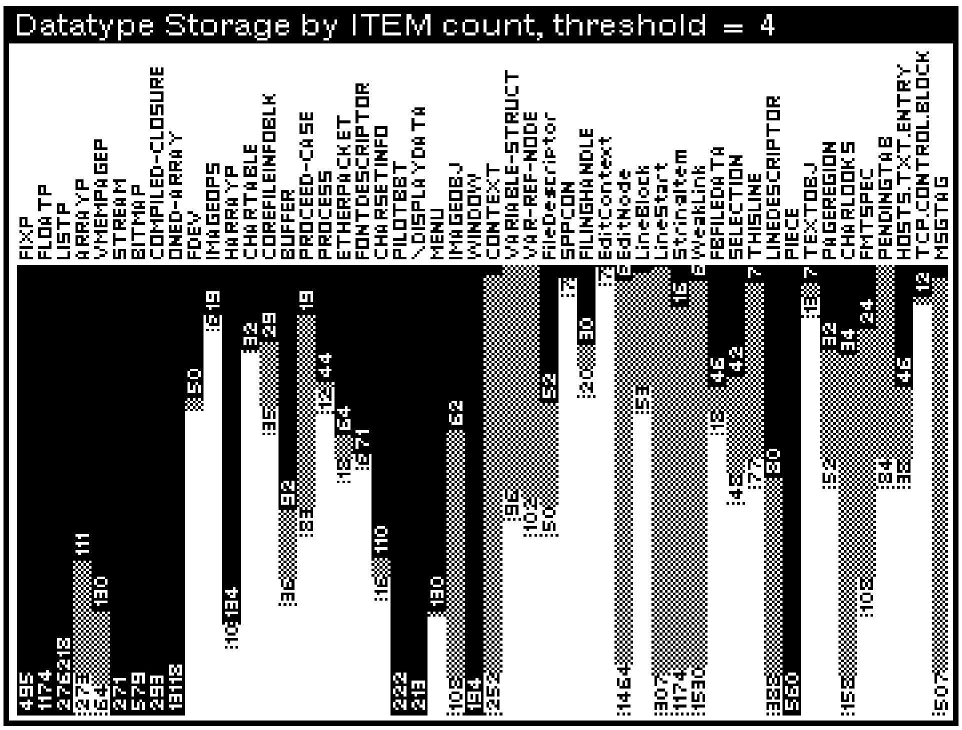
STORAGE

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

STORAGE implements a bar-graph version of the Lisp STORAGE function, providing a visual summary of the amount of storage allocated to each data type.

(SHOWSTORAGE *[PAGETHRESHOLD MODE ROTATION]*) [Function]

Displays the storage allocation of Lisp data types in bar graph format:



All the arguments are optional. *PAGETHRESHOLD* is the same as for the STORAGE function and defaults to **1**. *MODE* determines what to display and can be one of the following:

- **ITEM** The number of items of each type that have been allocated (the default mode).
- **PAGE** The number of pages allocated for each type.
- **BOX** The number of times each type has been allocated (see BOXCOUNT in the IRM).

The mode can be changed when the window is open by clicking with the *middle* mouse button. Clicking in the window with the *left* mouse button will update the window. When the window is redisplayed (using the standard window menu or REDISPLAYW) it will add new data types that have been defined since the window was last redisplayed.

For the **ITEM** and **PAGE** modes, the black part of the bar represents the number of items or pages currently *in use*. The gray part of the bar represents the number of *free* items or pages. The total length of the bar represents the *total* number of items or pages.

The *ROTATION* argument can be one of **NIL** (use the rotation of the SHOWSTORAGEFONT), **0** (labels from bottom to top on the right, bars grow to the left) or **90** (labels from left to right and bars grow down).

The display is controlled by the following global variables:

SHOWSTORAGEWINDOWSIZE [Variable]

The width or height (depending on the rotation) of the window, initially 275 (pixels). The bars truncate at the edge of the window; the window can be reshaped to put the longer bars in perspective.

SHOWSTORAGEIGNORE [Variable]

A list of data types to ignore. The information for the data types initially on this list is incorrect and/or their inclusion breaks the program.

SHOWSTORAGEDEFAULTTHRESHOLD [Variable]

The default threshold used when *PAGETHRESHOLD* is NIL, initially 1 (page).

SHOWSTORAGEPRIN2FLG [Variable]

Flag that causes PRIN2 to be used instead of PRIN1 when printing data type names (PRIN2 will include package names), initially NIL.

SHOWSTORAGEFONT [Variable]

The window font, initially one of Helvetica 5 through 10, i.e. the smallest that can be found when the file is loaded. The default font has a rotation of *90* degrees.

STYLESHEET

By: Tayloe Stansbury

Unsupported

INTRODUCTION

Stylesheets are collections of menus. These collections pop up all at once in a group. This group does not disappear until all menus in it have been dealt with, and the user signals that he is done.

Stylesheets are intended to be used in situations wherein the computer wants an answer to several related questions all at once. One example is font selection. To select a font, the user needs to specify font family (Classic, Modern, etc.), font size (8 point, 10 point, etc.), and font style (bold, italic, etc.). Rather than prompt for each of these parameters in succession, one could use a stylesheet to prompt for it all at once.

When the stylesheet pops up, it will shade (preselect) default selections (if provided) in each of the menus. The user can either decide that the default selections are OK, or change them to suit his taste. (The default selection mechanism can be used to convey the current state of something the user is trying to change with the stylesheet: for example, the current looks of the text with which the user is dissatisfied.)

When the user is finished, he hits the DONE button and the stylesheet disappears, and the final selections are returned. There is also a RESET button. This is useful if the user has mucked up his selections and would like to reinstate the default selections. Finally, there is an ABORT button that if selected returns NIL from STYLESHEET and is intended to provide the user with a convenient way of aborting the selection. Note: This means that NIL can be returned from a call to STYLESHEET.

Menus in a stylesheet can be set up to accept exactly one selection (like a normal menu), less than two selections, or any number of selections. Menus that need not be filled in (i.e., can accept zero selections) have an attached CLEAR button, which can be used to remove selections made in that menu. Menus that can have more than one selection have an attached ALL button, which can be used to select all the items in the menu.

HOW TO MAKE A STYLESHEET

To create a stylesheet, call

(CREATE.STYLE *Prop1 Value1 Prop2 Value2 ... PropN ValueN*) [Function]

CREATE.STYLE accepts an arbitrary number of property-value pairs. Properties currently recognised are

ITEMS [Style Property]

A list of menus. Most menu format parameters contained in menu records are honored by the stylesheet package. WHENSELECTEDFNs are, of course, ignored.

SELECTIONS [Style Property]

A list of menu items, each one corresponding to a menu in ITEMS. The specified selections will be shaded in the appropriate menu, and will be the default selections. If not specified or too short, it will be filled out with NILs (no selection).

NEED.NOT.FILL.IN [Style Property]

A list of T or NIL or MULTI, each one corresponding to a menu in ITEMS. T indicates that the corresponding menu need not be filled in and will be given a CLEAR button. MULTI indicates that the corresponding menu can have any number of selections and will be given both a CLEAR button and an ALL button. If the list is too short, it will be filled out with NILs. If a single T or NIL or MULTI is given instead of a list, it will be replaced by a list of Ts or NILs or MULTIs, respectively.

TITLE [Style Property]

The title that will be given to the stylesheet. If no title is specified, the stylesheet will not have a title bar.

ITEM.TITLES [Style Property]

A list of strings or atoms to serve as titles over the menus. Items without titles specified will not have titles.

ITEM.TITLE.FONT [Style Property]

A fontdescriptor or other font specification which determines the font item titles will be printed in. If NIL, titles will be printed in DEFAULTFONT.

POSITION [Style Property]

The screen position (of type POSITION) of the lower left-hand corner of the stylesheet. If position is not specified, the function STYLESHEET will prompt for the postion (using GETBOXPOSITION). STYLESHEET will modify positions as necessary to ensure that the entire stylesheet will be on the screen.

Stylesheets can be modified by calling

(STYLE.PROP *Stylesheet Prop Newvalue*) [Function]

STYLE.PROP always returns the old value of the specified property of the specified stylesheet. If Newvalue is provided (even if NIL), it replaces the old value. If not provided, the old value remains. (Just like WINDOWPROP.)

To use the stylesheet thus created, call

(STYLESHEET *Stylesheet*) [Function]

This returns a list of selections the user made from the stylesheet. (If a selection is returned as NIL, that indicates that no selection was made.)

One can determine in advance of displaying a stylesheet how big it will be. (This may help in determining a reasonable screen position for the stylesheet.) The relevant functions are

(STYLESHEET.IMAGEWIDTH *Stylesheet*) [Function]

and

(STYLESHEET.IMAGEHEIGHT *Stylesheet*) [Function]

They return the width and height, respectively, of the stylesheet in pixels.

AN EXAMPLE

The package is located in STYLESHEET and STYLESHEET.DCOM. To familiarize yourself with its workings, you might want to load it and try the following example:

```
(SETQ FONT.STYLE
  (CREATE.STYLE
    'TITLE "Please select a font:"
    'ITEM.TITLES '(Family Size Face)
    'ITEM.TITLE.FONT '(Modern 12)
    'ITEMS
      (LIST
        (CREATE MENU ITEMS ← '(Classic Modern Terminal))
        (CREATE MENU ITEMS ← '(8 9 10 11 12 14))
        (CREATE MENU ITEMS ← '(Regular Bold Italic BoldItalic)))
    'SELECTIONS '(Modern 11 Regular)
    'NEED.NOT.FILL.IN 'T]

(STYLESHEET FONT.STYLE]
```

SuperParentheses

By: Andrew J. Cameron, III (Cameron.pa@Xerox.com or cameron@cs.wisc.edu)

Most useful when used with: WHO-LINE (LispUsers)

This document last edited on Oct 19, 1987.

INTRODUCTION

This file, when loaded, creates a readtable (named "LISP[]") for use with CommonLisp which contain SuperParentheses, that is, the left square bracket (LEFTBRACKET syntax class) and right square bracket (RIGHTBRACKET syntax class) available in InterLisp. CommonLisp does not give these two characters their "usual" definitions, so as to allow users to easily give these character any macro/syntax definition they might desire.

This readtable will appear on, and can be selected via, the "Rdtbl" menu provided by the WHO-LINE LispUsers utility.

One can also access this new readtable with:

```
(IL:FIND-READTABLE "LISP[]")
```

These facilities obviate the need to store the readtable on a variable, as was done in an earlier version of this module.

INTERNALS

• The reading and writing of files using this readtable has not been tested or explored.

• SEdit is not too friendly to SuperParentheses.

• For more information, see Section 25.8.2 in both the IRM (InterLisp Reference Manual) and the Lyric Release Notes, and Section 22.1 in Steele (CommonLisp - The Language).

SYSTATS

By: Johannes A. G. M. Koomen
(Koomen.wbst@Xerox or Koomen@CS.Rochester)

This document last edited on: October 28, 1987

SUMMARY

SYSTATS provides a functional interface to system statistics such as PageFaults, DiskIOTime, etc. Statistics are maintained in objects of type SYSTATS. Functions are provided to fetch values from these objects, and to update the objects to reflect the current system state or to compute differences. This facility provides a Lyric alternative to the (undocumented) MISCSTATS functions in Koto.

DESCRIPTION

SYSTATSPROPS [Variable]

A list of statistics maintained by SYSTATS. Changing it does **not** alter SYSTATS behavior.

(SYSTATSPROP *prop fromstats*) [Function]

If *fromstats* is NIL, the internal SYSTATS object is updated and used. Retuns the value of the statistic named by *prop*, which must be a member of the variable SYSTATSPROPS.
Caveat: The value returned is a FIXP which is an element of the *fromstats* object and which, for the sake of performance, is reused during a SYSTATSREAD on the *fromstats* object. Note that there is an implicit SYSTATSREAD on the internal SYSTATS object if *fromstats* is NIL.

(SYSTATSREAD *intostats fromstats*) [Function]

If *intostats* is NIL, it is set to a newly created SYSTATS object. If *fromstats* is NIL, the internal SYSTATS object is updated and used. Copies system statistics from *fromstats* into *intostats*. Retuns *intostats*.

(SYSTATSDIFF *oldstats newstats difstats*) [Function]

If *oldstats* is NIL, the internal SYSTATS object is updated and used in its place. If *newstats* is NIL, the internal SYSTATS object is updated and used in its place. If *difstats* is NIL, it is set to a newly created SYSTATS object. Computes the statistics differences between *oldstats* and *newstats*, and places the results in *difstats*. Retuns *difstats*.

(CLOCKTICKS *interval timerunits*) [Function]

Returns the (machine dependent!) number of internal clock ticks over the interval. For instance, on the D'Lion, (CLOCKTICKS 2.5 'MINUTES) = 5211900.

TALK

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: Various editor and network protocol modules.

TALK allows users to hold conversations between machines across the Ethernet. TALK uses various *services* (TTY, TEdit and Sketch) and network *protocols* (NS and IP).

TALK FILES

Talk's services and protocols are now in separate files which may be loaded independently:

TALK	The main Talk module.

Services

TTYTALK	Simple text conversation between machines running Lisp, XDE and Viewpoint.
TEDITTALK	Uses TEDIT; allows the full capabilities of the TEdit editor in a conversation.
SKETCHTALK	Uses SKETCH; allows a conversation using the Sketch graphics editor.

Protocols

NSTALK	Uses COURIERSERVE (and optionally NSTALKGAP); allows XNS protocols.
NSTALKGAP	Used by NSTALK if the GAP Courier program has not been defined (by NSCHAT).
IPTALK	Uses TCP and TCPUDP; allows conversations using IP protocols.

Any Talk service can be used with any Talk protocol. The preferred order of loading is:

```
(FILESLOAD TALK TEDITTALK TTYTALK SKETCHTALK NSTALK IPTALK)
```

dropping out those services/protocols you do not use. Order of loading determines which services/protocols are tried first; the files may be loaded in any order to force different priorities.

USING TALK

(TALK *[USER.OR.HOSTNAME SERVICE PROTOCOL]*) [Function]

Starts a TALK session; *USER.OR.HOSTNAME, SERVICE* and *PROTOCOL* are optional. If not supplied, *USER.OR.HOSTNAME* is prompted for (either by menu or typein or both). If *SERVICE* and *PROTOCOL* are not supplied (the usual case) , TALK will figure out which to use based on what is available on the local and remote machines. The supported services and protocols are described below. The service and protocol used are indicated in the title bars of the TALK window.

TALK returns a process handle if the connection is successfully opened; it returns NIL if the user aborts out of the host/user menu and it returns an error message (as a string or list instead of breaking) if it cannot contact the remote host (for whatever reason). TALK can also be invoked from the background menu.

TALK MENU

The menu at the bottom of the TALK window (which is only active while the connection is open) contains the following items:

Disconnect Closes the TALK connection. This is equivalent to closing the TALK window, but leaves the window open in case you want to save and/or hardcopy part or all of the session.

RingBells Rings the bell on the remote workstation (if possible) and flashes the TALK window on the local one to indicate it has done so. This is useful if you have asked a person to hold and want to let them know you have returned.

Message Prompts for and inserts a *canned* message into the TALK stream. Useful if the phone rings and you want to ask the other person to hold with a minimum of time/effort. Messages can be added to the list, see the TALK.USER.MESSAGES variable below. The TALK window must have the keyboard in order to use this item.

ERROR MESSAGES

The TALK function will return one of the following error messages when it fails to start a session:

Host not found! It could not find host address for the host or user name specified.

Can’t connect to host! The remote workstation does not have the appropriate server loaded and/or running or does not have TALK loaded.

No answer from TALK service! A connection was made, but no one responded (intentionally or otherwise). A darkened TALK icon is left on the remote screen to log the connection attempt (unless TALK.GAG is non-NIL).

Unknown service type! An unknown type was given as the *SERVICE* argument.

No services available! The *SERVICE* argument was not supplied and it cannot find one.

Unknown protocol! An unknown protocol was given as the *PROTOCOL* argument.

No protocols available! The *PROTOCOL* argument was not supplied and it cannot find one.

Service and protocol errors may indicate additional files need to be loaded**.**

RECEIVING TALK

When your machine is contacted by another via TALK, the following icon will appear on your screen, ringing bells, flashing and showing the time, mode (service and protocol) and (when possible) the caller's identity:

If you button the icon with the left or middle buttons, a TALK session will begin. If you either close the icon or do not button it (it will go *dark* (invert) in about 15 seconds if not buttoned) the TALK connection will be refused. TALK connections are automatically refused if the TALK.GAG flag is non-NIL (settable using the subitem(s) of the TALK item in the background menu)**.** If the machine is in IDLE, TALK will wait twice the normal time out for the user to respond.

If you button a darkened (unanswered) TALK icon, it will try to reconnect you to the caller (after a mouse confirm). If a TALK connection comes in from someone who has already left an unanswered TALK icon on your screen, the icon will be reused.

TALK SERVICES

TEdit

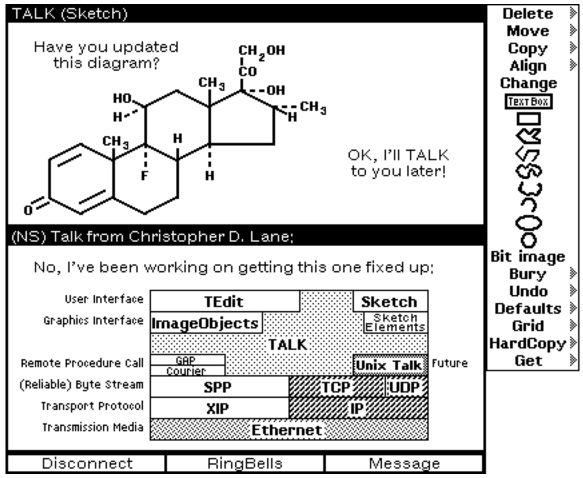
This service allows you to use the full capabilities of TEdit in your conversation, including: correcting mistakes anywhere in the document, changing character and paragraph looks, inserting ImageObjects, etc. Along with keyboard input, mouse selections and the caret are also visible to the remote user. The **GET** and **INCLUDE** commands in the TEdit command menu will load files into both the local and remote TEdit windows, so make sure the files are accessible to both. Similarly for fonts, if your workstation has to load a font from a server, the remote workstation must also have access to the font. Since the remote workstation may also need to load the font, you may experience communication delays. The TEdit service supports NS character codes and most of the 1108 and 1186 function keys.

TTY

This service is similar to the TEdit service except that the only supported feature is *backspace* (but not across lines). TTY is the only service that can talk with the Talk.bcd program in XDE or the TALK application (VPTalk.bcd) in Viewpoint. You *do not* need to know what type of workstation you are contacting when using any of the TALK programs.

Sketch

The Sketch service is built on the Sketch graphics editor:



TALK PROTOCOLS

NS

When NSTALK is loaded, TALK will accept as a host name anything that COURIER.OPEN will accept including an NS address or the name of a workstation registered in a Clearinghouse. Additionally, user names can be used if the address of the user's workstation is registered under the user's name in the Clearinghouse. The following function can be used to register a user and workstation correspondence in the Clearinghouse:

(CH.USER.WORKSTATION *USER WORKSTATION*) [Function]

Sets (or changes) the AddressList Clearinghouse property of *USER* (which must already be a name or alias in the Clearinghouse) to be the address of *WORKSTATION* (an NS address or name). If *WORKSTATION* is NIL, the function removes the AddressList property from *USER*. To use this function, you must be logged in (via `(LOGIN)`) as a System Administrator for *USER*'s domain.

One way to register users would be to go to the individual's workstation, login as the System Administrator and evaluate: `(CH.USER.WORKSTATION 'UserName \MY.NSADDRESS)`
Note that you cannot use the USERNAME function in this example since the (LOGIN) will change it.

NSTALK *does not* require or use NSCHAT, but they do share the Courier program *GAP*. If both NSTALK and the NS CHATSERVER modules are to be loaded, the CHATSERVER should be loaded *first* if possible. NSTALK is designed to allow other types of NSCHAT/GAP servers. The GAP server function determines which function to call using the service type requested (TTY = 5, TEdit = 6, Sketch = 7) and the entries on the association list GAP.SERVICETYPES which has entries of the form (ServiceNumber ServiceName ServerFunction). It is possible to have both NSTALK running and an EXEC server by adding appropriate entries to GAP.SERVICETYPES. If a GAP server already exists when NSTALK is loaded, it is made the default for all unrecognized service types.

Although NSTALK loads the COURIERSERVE LispUsers module you do not have to have a Courier server running to initiate an NS TALK connection, but you *must* have one running in order to receive an NS TALK connection.

IP (Interim)

When IPTALK is loaded, TALK will accept as a host name anything that DODIP.HOSTP will accept, including symbolic and numeric IP addresses. User names can be used by adding them as synonyms for local workstation hosts in the HOSTS.TXT file.

The current TALK IP interface is *only temporary* and will eventually be replaced by one which is compatible (for TTY service) with the TALK program which runs under BSD Unix; at that time, the allowable username format may be expanded to handle user@host. The current IP interface will probably not be compatible with the eventual, Unix-compatible one.

TALK VARIABLES

The following variables can be used to affect TALK's default behavior:

TALK.DEFAULT.REGION = (0 0 500 500) [Variable]

The LEFT and BOTTOM of this region determine where the (initial) TALK icon appears on the screen; the HEIGHT and WIDTH are the combined dimensions of the TALK windows (each uses half the HEIGHT). If this variable is set to NIL, then the icons start at (0 . 0) and the TALK window region is prompted for as needed.

TALK.USER.MESSAGES [Variable]

A list of menu items to put up when the MESSAGES item on the TALK menu is selected. Items on the list should return strings to be put into the TALK stream. If there is an entry of the form (GREETING "message") on this list, it will be printed automatically when a connection is opened.

TALK.GAG = NIL [Variable]

If non-NIL, causes the TALK server to automatically reject any TALK connections.

TALK.ANSWER.WAIT = 15 [Variable]

The number of seconds the TALK icon remains up before closing and aborting the connection.

TALK.HOSTNAMES = NIL [Variable]

A list structure containing hosts TALK has connected to along with the address used.

TALK.SERVICETYPES [Variable]

This list determines which services are tried and in what order. You only need to modify this if you wish to force an order other than the one determined by the order files were loaded or you wish to add or drop a service.

TALK.PROTOCOLTYPES [Variable]

This list determines which protocols are tried and in what order. You only need to modify this if you wish to force an order other than the one determined by the order files were loaded or you wish to add or drop a protocol.

KNOWN PROBLEMS

Talk

- Since TALK uses the Dove/DandeLion sound generator to help announce a connection, on other machines it is difficult for the user to detect connections being made during IDLE.

TTY Talk

- The TTY service cannot backspace beyond the left margin (unlike other implementations).

TEdit Talk

- Sometimes the local and remote TEdit windows will get out of sync as to what the current *looks* are; usually this is not serious.
- Page layout commands have not been implemented for the remote TEdit window; there are probably other commands that do not work either.
- ImageObject specific manipulations to ImageObjects already in the window do not get transmitted to the remote Tedit window.
- Inserting (other than keyboard input) into a pending delete does not echo correctly on the remote Tedit window.
- A large ImageObject inserted into the TALK window may not be seen by the remote user until some text is typed to force the remote window to scroll. The remote user may not see the ImageObject at all if it is larger than his window. These are both true of any TEdit window.
- User scrolling of the TEdit window will not cause scrolling of the remote TEdit window. System scrolling of the window (due to insertions and deletions) will be tracked in the remote window.

Sketch Talk

- When the TALK window is opened, some sketch menus will be created and then replaced. This is due to Sketch not allowing a user to specify both an existing window and an initial menu.
- When text (or a text box) is entered, only the initial character is seen in the remote window until the text is completed and the user buttons some other point in the window.
- Arrow heads do not show up at all on the remote sketch window.
- **Put** of a SKETCHTALK sketch gets into an infinite loop so temporarily you must copy the sketch items to another sketch if you wish to save them on a file.
- If you *sweep* a control point on a box past the other one (like sweeping one corner of a region past the other in RESHAPE), the remote box will not move identically.
- Since there are no functions to programmatically manipulate grouped elements the **Group** and **UnGroup** items have been disabled in the Sketch Talk window.
- For a small number of changes (text fonts, text box brushes and closed wire dashing), the entire remote sketch window is redisplayed to make the change visible.
- Setting the SKETCHINCOLOR flag to a non-NIL value will cause some operations to break.

TCPTIME

By: Christopher Lane (Lane@Sumex-Aim.Stanford.Edu)

Uses: TCP, TCPUDP

TCPTIME implements time client and server routines under TCP/IP and UDP/IP based on RFC868. The following are the user functions; the *PROTOCOL* argument refers to one of **TCP** or **UDP** and defaults to the value of RFC868.DEFAULT.PROTOCOL, initially **TCP**. All arguments are optional:

(RFC868.SETTIME *[RETFLG PROTOCOL]*) [Function]

Obtains the time from the network, similar to the \PUP.SETTIME and \NS.SETTIME functions. If *RETFLG* is non-NIL, the time is returned as an integer (as specified in RFC868), otherwise SETTIME is called and the new time is printed in the prompt window. Either TCP.TIME.HOSTS and/or UDP.TIME.HOSTS (see below) must be set before calling this function.

(RFC868.START.SERVER *[PROTOCOL ASCIIFLG]*) [Function]

Starts a network time server process for the specified (or default) *PROTOCOL* if one is not already running. The *ASCIIFLG* is discussed below.

(RFC868.STOP.SERVER *[PROTOCOL]*) [Function]

Deletes the network time server process for the specified (or default) *PROTOCOL* if one is running.

The following variables are used by the functions above:

RFC868.TIME.PORT = 37 [Variable]

Used to set the initial value of the protocol specific port variables when the file is loaded. Once the file is loaded, changing this variable has no effect, so it must be reset (if necessary) before loading the file, otherwise the protocol specific port variables should be reset directly. See TCP.TIME.PORT and UDP.TIME.PORT below.

RFC868.DEFAULT.PROTOCOL = TCP [Variable]

The default protocol to use when one is not specified.

BINARY & ASCII TIME FORMAT

Some network software implements the RFC868 standard by returning the printed (ASCII) representation of the time, rather than the binary representation as specified in the RFC. To work around this, the ASCIIFLG can be specified when starting a server to indicate that it should output the printed representation of the number. Similarly, when getting the time from the network, the following is used:

RFC868.ASCII.OSTYPES = (VMS) [Variable]

to decide based on the host's operating system whether to read the time as a binary or ASCII number. If this variable is set to NIL, the ASCII format is never used.

The ASCII format is currently only supported in the TCP protocol.

PROTOCOL SPECIFIC FUNCTIONS

(TCP.SETTIME *[RETFLG]*) [Function]

(UDP.SETTIME *[RETFLG]*) [Function]

Functions called by RFC868.SETTIME which can be called directly. The variables TCP.TIME.HOSTS and UDP.TIME.HOSTS must be set to use these functions.

(TCP.TIMESERVER *[ASCIIFLG]*) [Function]

(UDP.TIMESERVER) [Function]

Functions used by RFC868.START.SERVER. Can be used directly using ADD.PROCESS.

TCP.TIME.PORT = RFC868.TIME.PORT [Variable]

UDP.TIME.PORT = RFC868.TIME.PORT [Variable]

The ports to use in both the client and server functions.

TCP.TIME.HOSTS [Variable]

UDP.TIME.HOSTS [Variable]

Lists of host names and/or addresses (including broadcast addresses) to try to get the time from. Host are tried until one responds.

TCP.SETTIME.TIMEOUT = 10000 [Variable]

UDP.SETTIME.TIMEOUT = 10000 [Variable]

Length of time (in milliseconds) to wait for a host to respond to TCP.OPEN or UDP.EXCHANGE before trying the next one on the list.

TEdit-Close-On-Shrink

By: Nick Briggs (Briggs.pa@Xerox.com)

Uses: TEdit

This document last edited on August 4, 1987.

INTRODUCTION

TEdit has the unfortunate habit of keeping the file you are editing open when you shrink the TEdit window. For users of the FileCache this has the unfortunate sideeffect of preventing a newly saved file from being written out. In places where many people are editing files on file servers can lead to an excessive number of open files. TEdit-Close-On-Shrink attempts to handle some of the problem by persuading TEdit to close unmodified files when the edit window is shrunken.

USE

Load the module. It will install itself in the appropriate places (for the curious: that's `TEDIT.CREATEW`, `\TEDIT.CREATEW.FROM.REGION`, and `\TEDIT.REOPEN.STREAM`; and the shrink function `\TEDIT-CLOSE-ON-SHRINK`).

TEdit Dorado Keys

A set of convenience keys for the Dorado and TEdit.

INTERNAL

This package defines a number of meta-keystrokes as TEdit commands, providing much of the functionality of the 1186's expanded keyboard. An effort was made to keep the command set compatible with SEdit's for similar functions.

In a number of cases, the meta-lower-case and meta-upper-case commands are different--typically, the lower-case command turns some attribute (like boldness) on, and the upper-case command turns it off.

Here's the set of defined functions:

meta-u **meta-U**	UNDO
meta-f **meta-F**	FIND
metaa **meta-A**	ABORT (i.e., do a GET)
meta-s **meta-S**	SUBSTITUTE
ESC	REDO (note that this changes the default definition, which is EXPAND)
meta-x **meta-X**	EXPAND (also on control-X)
meta-n **meta-N**	Move to NEXT fill-in blank
meta-c **meta-C**	Change margins, rotating thru centered, left, right, just.
meta-b	Turn BOLD on
meta-B	Turn BOLD off
meta-i	Turn ITALIC on
meta-I	Turn ITALIC off
meta-=	Turn strike-thru on
meta-+	Turn strike-thru off
meta--	Turn underline on
meta-_	Turn underline off
meta-^	Superscript (or de-subscript)
meta-|	Subscript (or de-superscript)
meta-space	Return to default font/weight/slope/etc.
meta-?	Show current font in prompt window.
meta-(	Insert parentheses around the current selection
meta-"	Insert neutral double quotes (ASCII ") around the current selection
meta-'	Insert real double quotes (“ and ”) around the current selection

TEDITKEY

By: Greg Nuyens

Supported by: Jan Pedersen (Pedersen.pa@Xerox.com)

Uses: KEYOBJ, DLIONFNKEYS

TEditKey is a module that provides a keyboard interface to TEdit. On a Dandelion, the interface takes advantage of the special keys to the left, top, and right of the main keyboard. On a Dorado or Dolphin, a window mimicking the Dandelion function keys provides some of the same abilities.

The abilities provided include allowing the user to alter the *caret looks* (the looks of characters typed in) or the selection looks. These commands are given using the Dandelion function keys and/or metacodes. (Metacodes are keys typed while a meta key is held down. The default meta key is the tab key; to alter this see "User Switches" below.) Other metacodes and control codes move the cursor within the document (beginning/end of line, back/forward a character, up/down a line, etc.).

Thus, many of the special Dandelion keys are made to function in TEdit the way they are labeled. The following keys change their behavior once TEditKey is loaded.

CENTER modifies the justification of the paragraph(s) containing the current selection. If the selection was left justified, then hitting the CENTER key makes it centered. Hitting it again produces right and left justification.

BOLD boldfaces the selection. All other properties remain unchanged. Holding the shift key down while hitting BOLD will make the selection become un-bold.

ITALICS italicizes the selection. Shift-ITALICS is the opposite.

UNDERLINE underlines the selection. Shift-UNDERLINE is the opposite.

SUPERSCRIPT superscripts the selection by a constant amount. Any relative superscripts (or subscripts) are maintained. Thus if "X_i" is selected in "the set X_i is empty" then pressing the SUPERSCRIPT button produces "the set X^{X_i} is empty." See "User Switches" below for how to set the increment. Shift-SUPERSCRIPT is the same as SUBSCRIPT.

SUBSCRIPT is analogous to SUPERSCRIPT.

SMALLER decreases the font size of the selection. All relative size differences are maintained. E.g.,"this is bigger than that" produces "this is bigger than that." Shift-SMALLER (labeled LARGER) does the opposite.

DEFAULTS makes the selection have default looks. N.B.: The default looks can be set to the current caret looks by typing shift-DEFAULTS.

The above keys all affect the caret looks if the keyboard key is held down when they are hit. Thus holding down KEYBOARD and then hitting UNDERLINE makes the caret looks be underlined.

FONT changes the font of the selection or caret looks according to the following table (to alter this table see "User Switches" below):

1	Times Roman
2	Helvetica
3	Gacha
4	Modern
5	Classic
6	Terminal
7	Symbol
8	Hippo

Thus, to change the font of the selection to Classic, hold down FONT and hit 5. To change the caret font to Classic, hold down FONT (to signal the font change) and KEYBOARD (to direct the change to the caret looks) then hit 5. Note that this table is part of the menu displayed when the HELP button is pressed.

On a Dorado, middle-blank is the FONT key.

KEYBOARD applies any changes that occur while this key is down to the caret looks instead of the selection. On a Dorado, bottom-blank is the KEYBOARD key.

AGAIN invokes the redo facility in TEdit. A wide variety of operations can be repeated very simply by making a selection, performing an operation (for instance, an insertion), then picking a new selection and hitting the AGAIN key. The AGAIN key is an ESCape key, which acts as the TEdit REDO syntax class. (See page 20.22 of the *Interlisp Reference Manual.*)

OPEN opens a blank line at the current cursor position. OPEN is also used to type a linefeed outside of TEdit (for example to the function FILES?).

FIND prompts the user for a target string, then searches from the selection forward.

NEXT acts as the TEdit NEXT syntax class. (It goes to the next field to be filled in. These fields are marked as follows: >>text to be substituted<< .)

shift-NEXT transfers the TTY (which window will receive typed characters) to the next window which can accept typein. Thus one can cycle through the open text windows (mail windows, top level lisp windows, TEdit windows, etc.) without using the mouse.

EXPAND expands TEdit abbreviations. (See page 20.31 of the *Interlisp Reference Manual.*)

HELP displays a menu of the keybindings until a mouse key is clicked.

UNDO acts as the TEdit UNDO syntax class. Note that it still retains its TELERAID function as does STOP. There are TEditKey operations (such as Transpose Characters) that are implemented with multiple TEdit operations. Since TEdit will UNDO only single operations, it does not fully UNDO these operations.

RightArrow enters \, and **|** when shifted. (Recall that AGAIN is an escape key.)

MARGINS indents the margins of the paragraph selected. Shift-MARGINS exdents the margins. If the right margin is a floating margin, it is left unchanged. To control the amount by which the margins are moved, see "User Switches."

As well as the previous functions available on the Dandelion's special keys, the following functions are available on the standard keyboard (thus usable on the Dandelion, Dolphin, and Dorado). Each function is shown with the key that invokes it (in conjunction with the control (denoted ^) or meta (denoted #) key). Thus, for the sixth entry, holding down the metakey and hitting f (or "F") would move the caret one word forward. (To find out how to get a metakey see "User Switches" below.)

#/	defaults the caret looks
#=	queries caret looks
#9	smaller caret font
#0	larger caret font
^b	back character
^f	forward character
#b	back word
#f	forward word
^p	previous line
^n	next line
^a	beginning of line
^e	end of line
#<	beginning of document
#>	end of document
#s	select whole document
^k	kills line (delete from caret to end of line)
^o	opens line
^d	deletes character forward (also on shift backspace)
#d	deletes word forward (as always ^w deletes word backward)
^t	transposes characters
#[	indents paralooks. Also available on the MARGINS key
#]	exdents paralooks. Also available as shift-MARGINS
#j	justification change (same as CENTER key)
#u	uppercases selection
#c	capitalizes selection
#l	lowercases selection
#o	inserts object into document

#? shows keybindings (same as HELP)

#r restores the display

Note that the positions of any of these functions can be individually changed using TEDIT.SETFUNCTION (see page 20.30 of the *Interlisp Reference Manual*). For wholesale customization see "User Switches" below.

INTERRUPTS

Any operation can be aborted by typing the STOP key. This can be used to abort font changes, GETs, PUTs, etc. A stronger form of interrupt is available as shift-STOP, which prompts the user for a menu of processes to interrupt.

^G is available as a synonym for hitting the STOP key within TEditKey. Outside of TEdit, however, ^G will continue to have the meaning specified in the user's init file. This is often the HELP interrupt, which acts as shift-STOP.

Users who are accustomed to typing ^E as a soft interrupt should note that ^E moves to the end of the line. As discussed above, hitting the STOP key (or equivalently, typing ^G) accomplishes what ^E did.

Since ^H is defined to be the Backspace action in TEditKey, users cannot type ^A to erase characters even outside of TEditKey (Interlisp-D currently does not allow multiple backspace characters).

In addition to the changed functionality mentioned above (provided courtesy of TEditKey), the user should be aware of the following standard Interlisp-D/TEdit behavior:

SAME operates as a LooksCopy mode key. First make a selection. Now to copy the looks from some other text simply hold down the SAME key, then select the source for the looks. (Paragraph looks can be copied the same way, but by making the final selection while in the left margin. This is the standard way to select a whole paragraph in TEdit.)

MOVE and **COPY** act as mode keys for the selection mechanism. Thus the user can select the destination, then hold down the MOVE key and make a second selection. This selection will be moved (or COPY'd depending on the mode key used) to the (original) caret position.

CONTROL operates as a mode key to signal deletion. This means that holding down the CONTROL key and selecting some text will delete that text when the CONTROL key is released.

DELETE deletes the current selection when pressed.

DORADO EQUIVALENTS

Dandelion Key:	Equivalent key on Dorado:
OPEN	^o (or ^O)
SAME	META
FIND	finds item in TEdit menu
AGAIN	ESC
DELETE	DEL
COPY	SHIFT
MOVE	CTRL-SHIFT
PROP'S	META or LOCK depending on switches

NEXT	#n (or #N)
EXPAND	^x (or ^X)
HELP	#?
MARGINS	#[(unnest (which is shift-MARGINS on the Dandelion) is #])
FONT	top blank
KEYBOARD	middle blank
UNDO	bottom blank
STOP	^G
shift-**STOP**	#^S (intentionally difficult to type accidentally)

The function keys (CENTER, BOLD, etc.) are all available on the function key window brought up when TEditKey is loaded on a Dorado.

Note that the function key window can be rebuilt on a Dorado by selecting "Function Keys" in the default TEdit menu (obtained by buttoning in the title bar of a TEdit window).

USER SWITCHES

TEDITKEY.METAKEY The user must choose a metakey to make use of TEditKey. The value of the variable **TEDITKEY.METAKEY** is the name of the key that will be your metakey. For instance to make TAB (the default) your metakey, (SETQ TEDITKEY 'TAB) before loading TEditKey. (Note that even in the standard system, TAB is available as Control-I).

NOTE: METASHIFT (see page 18.9 of the *Interlisp Reference Manual*) is redefined to operate on TEDITKEY.METAKEY instead of on the bottom-blank key of the Dorado.

The operation of TEditKey is controlled by the following (INITVARed) variables:

TEDITKEY.LOCKTOGGLEKEY is the key that will be turned into a lock-toggle. If it is non-NIL, that key is set to act as a lock-toggle. Thus hitting this switches the case of the type-in. For those users who have removed the spring from their lock key, TEDITKEY.LOCKTOGGLEKEY is usually PROP'S. The action of LOCK is then made to be '(CTRLDOWN. CTRLUP) providing the user with a control key where LOCK is located and a lock toggle where PROP'S is located.

TEDITKEY.FONTS is an eight-element list of the fonts that are invoked by meta-1 through meta-8. The defaults are listed above.

TEDIT.DEFAULT.CHARLOOKS defines the looks that result when the DEFAULTS key is pressed or when default caret looks are requested. It is an instance of the CHARLOOKS datatype. To preset it, for instance, to TIMESROMAN 10 type the following to the Lisp top level.

(SETQ TEDIT.DEFAULT.CHARLOOKS (CHARLOOKS.FROM.FONT (FONTCREATE 'TIMESROMAN 10)))

However, a much simpler method is to select an instance of the desired looks and type shift-DEFAULTS**.**

TEDITKEY.VERBOSE if T (the default), the functions that modify the caret looks print feedback in the (TEdit) prompt window.

TEDITKEY.NESTWIDTH is the distance (in points) that the indent and exdent functions move the margins. Initially 36 points (0.5 inches).

\TK.SIZEINCREMENT is the amount (in points) which the LARGER function increases the selection (and conversely for SMALLER). Initially 2 points.

TEDITKEY.OFFSETINCREMENT is the amount (in points) which the SUBSCRIPT function raises the selection (and conversely for SUPERSCRIPT). Initially 3 points.

TEDITKEY.KEYBINDINGS is the list that controls the mapping of keys to functions for the functions that are common to the Dandelion, Dorado, and Dolphin. It consists of triples of function name, list of CHARCODE-style character specifications, and a comment describing what the function does. (The comments are used by the automated menu-building tools and their inclusion is encouraged.)

TEDITKEY.DLION.KEYACTIONS is the list that specifies the key actions of the non-Alto keys (to the left and right) on the Dandelion. It is the format acceptable to MODIFY.KEYACTIONS (see page 18.9 of the *Interlisp Reference Manual*).

TEDITKEY.DLION.KEYBINDINGS is the list specifying the functions to be tied to the characters generated from above. The keynames in the CAR of each element are comments. Note that TEDIT.DLION.KEYACTIONS and TEDIT.DLION.KEYBINDINGS must be coordinated (similarly for TEDITKEY.FNKEYACTIONS and TEDITKEY.FNKEYBINDINGS).

TEDIT.DLION.KEYSYNTAX is the list of syntax classes to be applied to the Dandelion keys.

TEDITKEY.FNKEYACTIONS is the list that specifies the keyactions of the function keys (center, bold, etc.).

TEDITKEY.FNKEYBINDINGS is analogous to TEDIT.DLION.KEYBINDINGS but for the function keys.

TEDITKEY.DORADO.KEYACTIONS are the keyactions unique to the Dorado (and Dolphin).

TEDITKEY.DORADO.KEYSYNTAX is analogous to TEDIT.DLION.KEYSYNTAX.

The previous variables in conjunction with the following functions specify the effect of TEditKey.

(TEDITKEY.INSTALL *readtable*) invokes the keyactions and bindings as specified by the above variables on *readtable*. (*Readtable* defaults to TEDIT.READTABLE).

(\TK.BUILD.MENU) is a function that automagically builds the help menu from the values of the above variables.

TEdit-Line-Numbering

By: Randy Trigg (Trigg.pa)

This document last edited on May 9, 1988

INTRODUCTION

TEDIT-LINE-NUMBERING enables the automatic conversion of a TEdit selection into multiple lines of specified width each ending in a carriage return and prefixed by a line number. For example, TEDIT-LINE-NUMBERING converted the following piece of transcript:

C: We have to be able to check that, within the memory I, I claim.

(1.0)

M: Check whether Tore is a graduate student? I think we can do that (.) I mean

C: Yea I know but more we have to be able to, within the memory somehow, recognize that (.) because of this constraint being in the memory, we have to check that the time matches Wednesday morning. We have to add this constraint to the time and see if (.) the time is not overconstrained.

----

into:

```
021  C:  We have to be able to check that, within the memory I, I claim.

022  (1.0)

023  M:  Check whether Tore is a graduate student? I think we can do that
024      (.) I mean

025  C:  Yea I know but more we have to be able to, within the memory
026      somehow, recognize that (.) because of this constraint being in
027      the memory, we have to check that the time matches Wednesday
028      morning.  We have to add this constraint to the time and see if (.)
029      the time is not overconstrained.
```

INTERFACE

(MAKE-LINE-BREAKS *TextStream WidthInInches Device StartLineNum LineNumDigits FirstLineParaLooks OtherLineParaLooks InsertExtraTabFlg InsertExtraCRFlg*)

TextStream should be a TEXTSTREAM of an open TEdit. Line breaks and line numbers will be inserted for the text in the current selection. WidthInInches is the number of inches wide the resulting lines will be when printed to Device. Device should be one of the litatoms DISPLAY, INTERPRESS, or PRESS. StartLineNum will be the line number used for the first line formed. LineNumDigits is the number of digits used to print the line numbers. FirstLineParaLooks should be a standard TEdit paralooks proplist for the first line formed from every paragraph. OtherLineParaLooks will be used for the remaining lines of each paragraph. InsertExtraTabFlg, if non-nil, will cause an extra tab to be inserted after the line number. InsertExtraCRFlg, if non-nil, causes an extra carriage return to be inserted between paragraphs. (If this last Flg is on, then FirstLineParaLooks and OtherLineParaLooks are probably equal.)

For example, the conversion depicted above was done with the following call (where SS is bound to an open textstream):

(MAKE-LINE-BREAKS SS 5 ’INTERPRESS 21 3 ’(LINELEADING 2 PARALEADING 5) ’(LINELEADING 2 PARALEADING 0) T)

The algorithm used is (pretty much) as follows: First, place a temporary marker at the end of the selection. For each paragraph in the selection, insert a line number LineNumDigits wide followed by a tab. Move one character at a time through the paragraph adding up STRINGWIDTHs until reaching WidthInInches (converted to appropriate units for Device). Then move back to last whitespace and insert a carriage return. Change the paralooks of the paragraph just formed according to FirstLineParaLooks. Insert the next line number followed by one or two tabs (depending on value of InsertExtraTabFlg) and continue adding up STRINGWIDTHs. When WidthInInches is reached, insert a carriage return and change the paralooks to OtherLineParaLooks. Continue in this manner till the end of the paragraph. If InsertExtraCRFlg is non-nil, then insert an extra carriage return. Continue with the rest of the paragraphs until encountering the marker at the end of the selection. Delete the marker and quit.

NOTES

It’s a good idea to have a couple of tabs set for the selected text, though MAKE-LINE-BREAKS will use the default tab setting if you don’t.

It tries to do proper measurement of embedded tabs, but this hasn’t been extensively tested.

BUGS

IT’S INCREDIBLY SLOW! This is because we use only calls available from the TEdit programmers interface. Things could be significantly sped up by walking the piece table like TEdit does when printing, but it’s alot of work to figure out how to do that and anyway I’d prefer that this tool only call advertised functions.

Doesn’t handle imageobjs.

TEDIT-PF-SEE

By:

Ron Kaplan

This document created in January 2022.

This tiny package adds alternatives to the PF and SEE commands that produce their output in scrollable read-only TEDIT windows rather than the unscrollable EXEC window. The new commands are tf (for t(edit)f(unction) and ts for t(edit)s(ee)

`tf FUNCTION (FILELIST) (REPRINT)` [command]

prints the definitions of `FUNCTION` that appear in the files in `FILELIST`, with a separate `TEDIT` allocated for each definition. If `FILELST` is not provided, then `WHEREIS` is used to locate the defintions, just as with `PF`. By default the definition characters are simply copied from the source file to the `TEDIT` stream, but if `REPRINT` is T the definition is read and then reprinted. This produces useful output for definitions that were not pretty-printed. Also, if `FUNCTION` is not provided, definitions for the last invocation will be reprinted.

`ts FILE (WINDOW)`

shows the contents of `FILE` in a scrollable read-only `TEDIT WINDOW`. `This uses the function` `TEDIT-SEE` (also used for the `FILEBROWSER See` command`)`, which interprets any font changes if `FILE` is a Lisp source file.

`TEDIT-PF-SEE` loads the `REGIONMANAGER` package, and the default regions for tf and ts are of type `PF-TEDIT` and `SEE-TEDIT` respectively. The function SET-TYPED-REGIONS can be used to predefine the regions where the output for tf and ts will appear.

TEdit-Process-Killer

By: Steve Bagley (Bagley.pa) and Randy Trigg (Trigg.pa)

This document last edited on Apr 23, 1987

INTRODUCTION

TEDIT-PROCESS-KILLER provides a simple interface to removing and restoring the process of a TEdit window. The processes of TEdit windows can be killed selectively or a TEDIT-KILLER process can be started to keep the total number of active TEdit processes at or near some threshold level. TEdit processes are automatically rebuilt when you button in their windows.

INTERFACE

(KILL-PROCESS-OF-TEDIT-WINDOW *WINDOW*) [Function]

kills the processes associated with the main window of WINDOW, and all of the attached windows. Each process is killed in such a way that the TEdit can be restarted. It is not an error to call this function on a TEdit whose process has already been killed.

(WITHOUT-TEDIT-PROCESS *WINDOW*) [Function]

returns T if this window does not have a process, because the process was killed by KILL-PROCESS-OF-TEDIT-WINDOW, NIL otherwise.

(RESTART-PROCESS-OF-TEDIT-WINDOW *WINDOW*) [Function]

restarts the TEdit processes for the main window of WINDOW and all attached windows if the processes have been killed by KILL-PROCESS-OF-TEDIT-WINDOW.

(START-TEDIT-KILLER) [Function]

starts up a process called TEDIT-KILLER which wakes up at regular intervals to kill off the least recently used TEdit processes. There are two global vars available to the user to affect its operation:

TEDIT-PROCESS-LIMIT [Variable]

Defaults to 10. The preferred threshold of running TEdit processes. Every time TEDIT-KILLER wakes up, it kills off enough TEdit processes to bring the total down to this limit.

TEDIT-KILLER-WAIT-TIME [Variable]

Defaults to 10000. The time in milliseconds between wake-ups of TEDIT-KILLER.

(STOP-TEDIT-KILLER) [Function]

kills any running TEDIT-KILLER process.

NOTES

In order to force a TEdit to be killed off when shrunk, simply do

```
(WINDOWADDPROP <Win> 'SHRINKFN (FUNCTION KILL-PROCESS-OF-TEDIT-WINDOW))
```

and, if you like,

```
(WINDOWADDPROP <Win> 'EXPANDFN (FUNCTION RESTART-PROCESS-OF-TEDIT-WINDOW))
```

BUGS

We don’t kill lafite sendmessage processes.

TEKTRONIX

By: Jim Blum (Unsupported)

Last Modified: James Turner (Turner:Lexington:Xerox)

TEKTRONIX 4010 EMULATOR

FILES: TEK4010CHAT, TEK4010

This document last edited on: 9-May-88 23:52:18

There are two LispUsers Modules, TEK4010 which takes an INSTREAM and OUTSTREAM as arguments and emulates a TEKTRONIX 4010 storage tube terminal, and a version called TEK4010CHAT which works with CHAT. The details on how and what a TEKTRONIX 4010 terminal does are described in a TEKTRONIX 4010 users manual. The CHAT program is described in the Interlisp Reference Manual. This document will point out the differences from the default DM2500 emulated terminal which CHAT uses, and the relevant issues unique to running the TEK4010 emulator under Interlisp.

TEK4010 is called as follows:

(TEK4010 *INSTREAM OUTSTREAM*) [Function]

where OUTSTREAM must be a displaystream (or window). This version does not fully support the TEKTRONIX 4010 terminal. Specically, the graphic input mode (which displays the crosshair and waits for a key to be typed), and does not support the double column text mode. This version supports a limited scaling feature. Both the TEKTRONIX 4010 X and Y coordinates are divided by one global integer called TEKPTSPERPOINT which can be set(q) by the user. No attempt is made to scale the text or line spacing (leading) in this version, nor does reshaping the window automatically change TEKPTSPERPOINT.

The CHAT version (TEK4010CHAT) unlike the DM2500 emulator, runs in non-scroll, paint mode, with the right margin set at the width of the window. Although not elegant, this is how the TEK4010 terminal works and there may be cases where users are dependent on its inherent mode of operation. The visible implications of this on INTERLISP are: The screen is not cleared or scrolled when a linefeed is received on the last line. Graphics and text are overlayed (OR'd) using the PAINT mode in BITBLT, since selective areas on a storage tube cannot be erased. Graphics are scaled to the window size and so is the position of the second margin, but the characters in the font (GACHA 10) are not scaled. Scaling is changed when the CHAT window is reshaped. In order to get accurate positioning of both the text and graphics, the Interlisp window size should match the TEK4010 screen resolution which is 1024 horizontally by 768 vertically.

In order to run the TEK4010 emulator under CHAT, the following recipe must be followed:

1. Make sure you have the latest version of CHAT loaded.

2. Load the TEK4010 emulator module, ie, (LOAD ’TEK4010CHAT.LCOM)

3. Edit the variable CHAT.DISPLAYTYPES (ED ’CHAT.DISPLAYTYPES) to add one or more lists of the form (host number TEK4010) where host is the name of the host that you want to chat to with this emulator. Use the number 4010 for the number field which would be used by the CHAT.SETDISPLAYTYPE function. Add as many entries as there are hosts you want to use the TEK4010 emulator with.

4. Then bug CHAT in the background MENU and select or enter the host that you want to chat to.

TILED-SEDIT

By: Johannes A. G. M. Koomen
(Koomen.wbst@Xerox or Koomen@CS.Rochester)

This document last edited on: September 23, 1987

SUMMARY

TILED-SEDIT is a facility for automagically positioning SEdit windows according to a specified pattern. SEdit windows appear in any of the four corners of the screen, with overlapping windows slightly offset so they can still be brought to top (by clicking on them). Users can specify which corners, in what order, how thick a margin around the screen, and the size of the offset.

DESCRIPTION

(TILED.SEDIT.RESET *Tiling-Order XShift YShift Screen*) [Function]

If *Tiling-Order* is NIL, this resets the SEdit window tiling facility, and SEdit reverts back to its old behavior (i.e., prompting for a window region). Otherwise *Tiling-Order* should be either T or a keyword or an arbitrarily long list of keywords from the following set { :TL :TOP-LEFT :TOP.LEFT :TOPLEFT :BL :BOTTOM-LEFT :BOTTOM.LEFT :BOTTOMLEFT :TR :TOP-RIGHT :TOP.RIGHT :TOPRIGHT :BR :BOTTOM-RIGHT :BOTTOM.RIGHT :BOTTOMRIGHT }. If *Tiling-Order* is T, the list '(:TL :BL :TR :BR) is assumed. SEdit will place new windows in the corners specified by *Tiling-Order* (which is indefinitely repeated if necessary).

If a new SEdit window would overlap an existing SEdit window, the new one is offset by *XShift* pixels right and *YShift* pixels down. *XShift* and *YShift* default to 15. Tiled.SEdit will compute the tile size and placement on the basis of the region *Screen* such that you can go three times through the default four corner loop before the right or bottom windows start crossing the edge of *Screen*. If *Screen* is neither a region nor a fixp, *Screen* defaults to 25. If *Screen* is a fixp M, *Screen* is assumed to be (CREATEREGION M M SCREENWIDTH-M SCREENHEIGHT-M). The default setting leaves room enough for a scrollbar on the left and the bottom.

Invoking TILED.SEDIT.RESET with a non-NIL *Tiling-Order* will cause all currently open SEdit windows to be repositioned according to *Tiling-Order*.

EXAMPLES

(TILED.SEDIT.RESET T) [Function]

This is executed when you load TILED-SEDIT. It provides for automatic SEdit window creation in the corners TopLeft, BottomLeft, TopRight, BottomRight, TopLeft, BottomLeft, ... Each time around the loop windows are shifted 15 pixels to the right and downward. A 25 pixels margin is preserved at the left and bottom edge of the screen.

(TILED.SEDIT.RESET :TL) [Function]

This causes SEdit to create windows in the TopLeft corner only. Each new window is shifted 15 pixels to the right and downward. A 25 pixels margin is preserved at the left and bottom edge of the screen.

(TILED.SEDIT.RESET '(:TR :BR) NIL 35) [Function]

This causes SEdit to create windows in the TopRight and BottomRight corners only. Each time around the two corner loop windows are shifted 15 pixels to the right and 35 pixels downward. This has the advantage that the title of each SEdit window remains visible, but the disadvantage that each window is smaller. A 25 pixels margin is preserved at the left and bottom edge of the screen.

CAVEAT

TILED.SEDIT.RESET is independent of SEDIT.RESET. It will not invoke SEDIT.RESET, nor does it require that all SEdit windows are closed prior to invocation. It is strictly used for controlling the window tiling.

TKDORADO

By: Mike Dixon (MikeDixon.pa)

INTERNAL

TKDorado: TEditKey for the Dorado keyboard

TKDorado makes the full range of TEditKey commands available from the Dorado keyboard. To do so it adds TEditKey bindings for a number of meta-control key combinations:

`Meta-Control-`**`V`**`:`	set to default (vanilla) looks
`Meta-Control-`**`B`**`:`	bold on
`Meta-Control-`**`N`**`:`	bold off
`Meta-Control-`**`I`**`:`	italics on
`Meta-Control-`**`O`**`:`	italics off
`Meta-Control-`**`D`**`:`	overbar on
`Meta-Control-`**`F`**`:`	overbar off
`Meta-Control-`**`G`**`:`	strikethru on
`Meta-Control-`**`H`**`:`	strikethru off
`Meta-Control-`**`J`**`:`	underlining on
`Meta-Control-`**`K`**`:`	underlining off
`Meta-Control-`**`[`**`:`	smaller font
`Meta-Control-`**`]`**`:`	larger font
`Meta-Control-`**`^`**`:`	superscript
`Meta-Control-`**`_`**`:`	subscript
`Meta-Control-`**`C`**`:`	center/justify/left this paragraph

The functions which change character looks operate on the current selection, if any characters are selected, and otherwise change the caret looks.

The new key bindings are also added to the help menu.

In addition, TKDorado rebinds the ESC key to be REDO, rather than Expand (which is still available as Control-X), and closes the DLion keys window (if TEditKey has left it lying around).

TMAX

Chapter [Chapter] Introduction

TMAX stands for Tedit Macros And eXtensions and it enhances TEdit by providing a convenient way to do things such as numbering, indexing, creating a table of contents, and more. At SUMEX these sort of operations are done with Scribe[Note#]. Scribe is a powerful document preparation language but it consumes all together too many cycles on our mainframe. Furthermore with Scribe you must hardcopy your document to see what it looks like. If you want to change the format, you must add the appropriate commands to the Scribe input file, run Scribe again, and hardcopy the output file. This sort of "batch processing" wastes both yours and the mainframe's time. TEdit is a WYSIWYG (What You See Is What You Get) text editing and formatting system. You see what your document will look like while you are creating it.

TMAX makes no attempt to mimic Scribe in TEdit nor does it have any facility to translate a Scribe `.MSS` (i.e. source) file into a TEdit file. Rather it implements some of the more commonly used features of Scribe in TEdit. Currently there are four main areas in TMAX; indices, numbering, endnotes, and forward and backward references. The TMAX features described here plus the editing and formatting features already available in TEdit make it an attractive alternative to Scribe. For more information on TEdit, please read the first part of the **Text Editing** section in **The Lisp Library Packages Manual**.

All the features described here are merely additions to a TEdit document. TMAX does not alter your text in any way. When you invoke one of these features, TMAX inserts a "special character" (i.e. an Image Object[Note#]) into your document at the current cursor position. These "special characters" may appear to be strings but they are really just single characters. This means you can delete any TMAX feature you add by simply deleting the corresponding "special character" just as you would delete any other character. The features described in this document are also used throughout this document. Rather than including pictures of each menu (and there are lots of them), we have decided to show you how to pop-up these menus yourself. This document is both an explanation and example of how to use TMAX. We suggest you read this document under TMAX/TEdit and try these features as you read about them. This document was written assuming you are reading it in a TEdit window.

Chapter [Chapter] Menus

When you first load TMAX.DCOM, a new item called **TMAX Menu** is added to the main TEdit pop-up menu. Since you probably used this menu to **Get** this document, you may have already seen this new item. Buttoning **TMAX Menu** brings up the TMAX menu which is attached to the top of your TEdit window. Now you can invoke the TMAX features by simply buttoning items in this menu. You can remove this window by right buttoning the mouse in its title bar and selecting `Close` (just as you would remove any other TEdit menu). The small window just above this window is called the "prompt window". TMAX uses this window both to prompt you for text input and to report current values and status. Please pop up the TMAX Menu now by pointing the mouse at the black bar above this window, left or middle buttoning it, and selecting **TMAX Menu**.

There are three types of fields in the TMAX menu; labels, values, and commands. You select items by pointing the mouse at the item and pressing either the left or middle button.

- [Item] The fields on the far left of each line (Miscellany:, References:, etc.) are simply labels that specify the nature of the items on the rest of that line. You cannot select these label fields.
- [Item] There are eight items that are used to set values. Five of the items are toggle switches, two define output filenames, and one pops up a menu of possible settings.
  - [Subitem] The five toggle switches are ***Set AutoUpdate***, ***NGroup Menu***, ***Text Before***, ***Text After***, and ***Manual Index***. They are all in a bold italic font to distinguish them from other items. These 2-way toggle switches appear in normal video (as they are above) when they are off and in reverse video when they are on. Buttoning these items complements their current setting.
  - [Subitem] **TOC Filename:** and **Index Filename:** are used to define the name of the Table-Of-Contents and Index file respectively. When these items are buttoned, the cursor appears to the right of the colon and TMAX waits for you to specify a filename. On a **Get** these items are defaulted to the name of the input file with the extensions `.TOC` and `.INDEX`.
  - [Subitem] The last value item is **Reference By** and the light face field following it is its current setting. Buttoning this item pops up a menu of possible settings.
- [Item] All the other items (everything except labels and values) are commands. Buttoning these items invokes the corresponding TMAX feature. These items turn to reverse video while they are active and then return to normal video when they finish.

Many of the TMAX features use pop-up menus. Buttoning the mouse outside any pop-up menu is equivalent to no selection and will make that menu disappear. Whenever TMAX prompts you for a new value, it always displays the current value in the prompt window.

We will now discuss each item in the TMAX menu in detail. Since many of the items interact with other items, it is difficult to find a starting point. Instead we have described the TMAX menu row by row and left to right within each row. We suggest you first peruse this document to get a general idea of the features and then focus on the areas that interest you.

Chapter [Chapter] Miscellany

[Section] Update

When you use TMAX to do some sort of numbering, TMAX inserts a "marker" rather than the actual number. It does this for speed since it is much faster to insert a marker rather than figure out the corresponding numeric value each time. Buttoning **Update** will convert all these markers to their corresponding (and consecutive) numeric values. In addition, any References (see chapter <Chapter/Value>) to these numbers will be updated. Finally, if the Endnotes (see chapter <Chapter/Value>) have already been inserted, **Update** will reinsert them but this time with the actual Endnote numbers. Currently the only numbering constructs in TMAX are Endnotes, Number Groups, and References to them. Please button **Update** now and watch all the changes.

[Section] Undo Update

Buttoning this item undoes everything that **Update** does. That is it converts all the Endnote and Number Group numbers back to their corresponding markers along with any References to these numbers. If the Endnotes have already been inserted, **Undo Update** will reinsert them but this time with the Endnote numbers replaced by their markers.

[Section] Set AutoUpdate

When this toggle switch is on, all numbering markers are immediately updated to their corresponding numeric value as they are inserted. Note that it only updates inserted markers; it does not automatically update any markers in a TEdit file that you load via **Get**. We do not encourage setting this switch for large documents. This switch causes TMAX to check every number it generated in the entire document whenever a new number is inserted. For large documents this could take some time.

[Section] Current Date/Time

Buttoning this item inserts the current date into your document. For example, the date enclosed in the following parentheses (April 22, 1987) was inserted by buttoning **Current Date/Time**. Middle buttoning this inserted date pops up a Date/Time menu. If you'd like to see this menu, go ahead and middle button the date in the parentheses. This menu allows you to change the format of the date/time, replace the date with the time, or update an old date/time to the current date/time. The font is the same as the font currently in effect in your TEdit document when the date/time was inserted.

Chapter [Chapter] References

These commands allow you to reference Endnotes and Number Groups by either their numeric value or by the page number they appear on. A Reference is simply an association between a "Tag" and a reference to that Tag. To assign a Tag, middle button the Endnote or Number Group marker (or its numeric value if it has already been updated). A menu will pop up and one of the items will allow you to define a Tag for the Endnote or Number Group. If the Endnote or Number Group is already tagged, the menu will allow you to change or delete the Tag. All the Tags in a document must be unique and TMAX will not allow you to create a Tag that is already defined. If you COPY a tagged Endnote or Number Group within the same window, TMAX will remove the Tag on the copied object. If you COPY a tagged object to another TEdit window, TMAX will preserve the tag assuming that Tag name isn't defined in the other TEdit window. The font of a Reference is the same as the font currently in effect in your document regardless of what font is used to display the Endnote or Number Group.

As an example, we have tagged both the Endnotes chapter (a Number Group on page <Chapter/Page>) with "EN Chap" and the Endnote at the end of this sentence with "EN Note"[Note#]. Now we can reference the Endnotes chapter as <Chapter/Value> and the previous Endnote as <Note#/Value>. Also the page number of the Endnotes chapter above was generated by a Reference to "EN Chap" by page instead of by value. You will have to button **Update** to convert the "`<Chapter/Value>`" and "`<Note#/Value>`" to their respective numeric values.

Of course, if ***Set AutoUpdate*** was on, the "<Chapter/Value>" and "<Note#/Value>" would be converted as soon as they were inserted.

[Section] Reference

TMAX will prompt you for a Tag name when you button this item. A bare carriage return cancels this command. It doesn't matter if the Tag you specify is defined yet or not. If the Tag is not defined, the Reference marker is "<Reference Tag/Type>" where Tag is the Tag name and "Type" is either Value or Page. In this case the Tag name is embedded in the marker. If the Tag is defined, the Reference marker is either <Note#/Type> for Endnotes or the Number Group marker enclosed in angle brackets (e.g. <Chapter/Type>). In this format there is no indication of what the Tag name is. If you middle button a Reference marker (whether it is updated or not), TMAX will display the corresponding Tag in the prompt window. All Reference markers that have not been updated are enclosed in angle brackets (i.e. < and >) to distinguish them for normal text.

[Section] Previous References

Buttoning this item brings up a menu of all the defined Tag names in alphabetical order. You can then create a Reference to one of these Tags by simply buttoning the corresponding Tag in this menu. Although it doesn't matter if you define the Tags or References to these Tags first, we suggest you define the Tags first. Once the Tags are defined you can use this command to easily create the References to these Tags. If you button **Previous References** now you will see all the Reference Tags defined in this document.

[Section] Reference By

This item allows you to select whether you want References by numeric value (of the Endnote or Number Group) or by the page number (on which the Endnote or Number Group appear). Buttoning this item pops up a menu with three items; Ask, Value, and Page. If you select Value or Page then all References you add will be of the type you specified. If you select Ask then TMAX will prompt you for the type every time you make a Reference. As you can see in the TMAX menu above, the default Reference type is by Value.

There is one minor inconvenience with doing forward references by page. You must do a dummy hardcopy and then button **Update** before the real hardcopy. The reason is TEdit only knows the current page number while it is hardcopying a document. As TMAX encounters Endnotes and Number Groups, it saves their corresponding page numbers. If you reference an Endnote or Number Group that TMAX hasn't seen yet then it can't possibly know the corresponding page number. In this case, TMAX will change the Reference marker to the Endnote's or Number Group's numeric value followed by "/Page" both enclosed in angle brackets (e.g. <1.2/Page>) to indicate that the page number is not known yet. This only applies to forward references by page. You don't have to do a dummy hardcopy if you are referencing by value or doing backward references by page.

Chapter [Chapter] Endnotes

These commands allow you to generate a numbered list of notes. Endnotes are like footnotes except the numbers and corresponding text appear at the end of the document rather than the bottom of the page. (TMAX does not support footnotes yet.) The Endnote numbers are consecutive and always start at 1. If you add an Endnote in the middle of a document, all the following Endnote numbers will be adjusted automatically. Suppose for example you have three Endnotes and you add another between Endnotes 1 and 2. The new Endnote becomes 2 and the Endnotes that were formerly 2 and 3 are now 3 and 4. This will happen when you button **Update** (or immediately if ***Set AutoUpdate*** is on). The font of the Endnote numbers, text, and title line is determined by the **Set Style** command and the default font is Gacha 10 Standard.

If you middle button an inserted Endnote marker, a menu will pop up allowing you to define a Tag for this Endnote marker or edit the text associated with this Endnote marker. If this Endnote already has a Tag, this menu will give you the option of changing, deleting, or displaying the Tag or editing the Endnote text. If you choose to edit the Endnote text, you will be prompted to open another TEdit window where the text will be displayed. When you are done editing the text, move the mouse to the title bar (i.e. the thick black bar at the top of the new TEdit window) and left or middle button the mouse. A menu will pop-up giving you the option of saving or aborting the changes. There is a sample Endnote at the end of this sentence you can play with[Note#].

[Section] Endnote

Buttoning this item inserts an Endnote marker at the current cursor position. The Endnote marker is a superscript "Note#". You will also be prompted for the text associated with this Endnote.

[Section] Insert Endnotes

This item inserts the title line "Notes" followed by the Endnote numbers and their corresponding text after the last line in your TEdit document. If you have already inserted the Endnotes and button this command again, TMAX will delete the old Endnotes and then reinsert them. It does this in case you have added or deleted any Endnotes since the last time they were inserted.

[Section] Delete Endnotes

Buttoning this item undoes what **Insert Endnotes** does. It deletes the title line "Notes" and the Endnote numbers and text that follow from the end of your document.

If you already inserted the Endnotes and wish to add more text at the end of your document, you ***must*** first delete the Endnotes with this command. If you don't, the additional text at the end of your document will be deleted when the Endnotes are reinserted. Suffice it to say this is due to the way Endnotes are implemented rather than a bug in TMAX. A future version of TMAX will fix this awkwardness.

[Section] Set Style

Buttoning this item pops up the Endnote Fonts menu. This menu has three items; `Number`, `Title`, and `Text`. You can use this pop-up menu to change the font of the Endnote numbers, title line, and text. The title line is always "Notes". When you select one of the items from this menu, the current font of that item will be displayed in the prompt window before TMAX prompts you for the new font.

Chapter [Chapter] Number Groups

The Number Group (NGroup) commands allow you to number any arbitrary objects in your TEdit document. You define the hierarchy, font, and format of each NGroup member and then insert them wherever you want something numbered. The NGroup numbers are consecutive and start at 1 by default although you can change the starting value. This allows you to number individual pieces of a document without having to load the entire document. Some NGroup values depend on other NGroup values. For example, the sections of this document each begin with the corresponding chapter number. Each time the chapter number changes, the section number is reset to its starting value. You can nest NGroups as deep as you like. If you insert a new NGroup member in the middle of a document, all the following NGroup members that depend on this new member will be adjusted when you button **Update** (or immediately if ***Set AutoUpdate*** is on). This makes it trivial to number (and renumber) things. All you do is insert an NGroup member wherever you want something numbered and TMAX takes care of all the rest. The default NGroup font is Gacha 10 Standard.

[Section] NGroup Menu

When this toggle switch is on, the NGroup menu "graph" is displayed in a window attached to the top of the TMAX menu window. The Number Group graph is a tree structure showing the hierarchy and order of all the Number Group members. If you haven't buttoned ***NGroup Menu*** yet, please do so now.

[Section] New NGroup

This item allows you to create new Number Group members. When it is buttoned, TMAX first turns on ***NGroup Menu*** if it was off so you can see the NGroup graph. It then prompts you for the name of the NGroup member. Finally TMAX prompts you for the parent of this new NGroup member by popping up a menu of the known NGroup names and you select the parent from this menu. By parent we mean the node to the immediate left in the NGroup graph. If you don't select any parent (i.e. button the mouse outside the pop-up menu) the new NGroup becomes a "top level" node. A top level node is one whose parent is the boxed **NGroups** (at the far left). The NGroup graph is always built from left to right. Since the first NGroup member defined will have no parents, TMAX just adds this member to the graph. You can create as many NGroup members as you like even if you don't use them all. As long as there is at least NGroup in you document, the entire NGroup graph will be saved/restored over a **Put**/**Get**.

[Section] Text Before

When this toggle switch on, you will be prompted for a preceding text string each time you insert a NGroup member. This text string becomes part of the inserted NGroup marker. You can use this string as a heading for the chapter, section, figure, example, etc. you have numbered. The text string is always printed in the same font as the corresponding NGroup member and TMAX always inserts one space between the preceding text string and the NGroup marker. If you would like more spaces, you must put them in the ***Text Before*** string yourself. Any tabs in this string are automatically converted to spaces. The default ***Text Before*** string is the name of the NGroup. In this document all the Chapter NGroup markers were inserted with ***Text Before*** on and the Section NGroup markers were inserted with ***Text Before*** off.

[Section] Text After

When this toggle switch on, you will be prompted for a succeeding text string each time you insert a NGroup member. This text string becomes part of the inserted NGroup marker. You can use this string as a heading for the chapter, section, figure, example, etc. you have numbered. The text string is always printed in the same font as the corresponding NGroup member and TMAX always inserts one space between the NGroup marker and the succeeding text string. If you would like more spaces, you must put them in the ***Text After*** string yourself. Any tabs in this string are automatically converted to spaces. There is no default ***Text After*** string. In this document all the Chapter and Section NGroup markers were inserted with ***Text After*** on.

[Section] Changing the default Font and Format

Consider the Number Group graph in this document. The boxed **NGroups** at the far left is a special node and buttoning it does nothing. The order of the nodes in a branch is important but the order of the branches themselves isn't. If you would like to see a more complex NGroup graph then just add some more members to this graph. Don't worry; adding extra NGroup members doesn't affect anything.

When you create NGroup members by buttoning **New NGroup**, you are actually defining "prototype" NGroups. The font and format of these prototypes determine the font and format of the NGroups you insert into your document. To change the font and/or format of a prototype NGroup member, simply point the mouse to the member name in the Number Group menu graph and ***middle*** button it. A menu will pop up with four items; `Change Font`, `Show Font`, `Change Format`, and `Show Format`. Buttoning `Show Font` or `Show Format` will display the font or format of the selected prototype NGroup in the prompt window.

Buttoning `Change Font` or `Change Format` allows you to change the font or format of the selected prototype NGroup. This is a *global* change. If you change the font/format of a prototype NGroup, TMAX will apply that change to every occurrence of that NGroup in your document. Before TMAX prompts you for the new font/format, it always shows you the current font/format in the prompt window. `Change Font` and `Change Format` both have subitems that allow you to change part of the font/format without changing anything else.

The subitems for `Change Font` are `Family`, `Size`, and `Face`. If you button one of these subitems, TMAX will pop up the corresponding menu and only that part of the font will be changed. If you button `Change Font` instead, TMAX will prompt you for all three values via pop-up menus.

There are seven subitems for `Change Format`. You can change any part of the format by selecting one of these subitems. If you button `Change Format` instead, TMAX will prompt you for all seven values. Six of the values use pop-up menus and the other requests input in the prompt window. The seven parts of the prototype format are:

[Sect Cntr] `Delimiter Before`

This is the delimiter that precedes the NGroup. TMAX pops up a menu of commonly used delimiters. You can select one of these or select `Other` in which case TMAX will prompt you for a delimiter string in the prompt window. The default `Delimiter Before` is a null string.

[Sect Cntr] `Display Type`

This is how the NGroup's numeric value is displayed. TMAX pops up a menu of the various ways a NGroup can be displayed. These ways are as an Arabic numeral, a null string, or an upper/lowercase letter or Roman numeral. The default is as an Arabic numeral.

[Sect Cntr] `Delimiter After`

This is the delimiter that follows the NGroup. It uses the same mechanism as the `Delimiter Before`. The default `Delimiter After` is a period (i.e. ".").

[Sect Cntr] `Abbreviate Level`

Normally a NGroup's value is the concatenation of all its parents values plus its own value. This subitem allows you to specify how far up the NGroup branch to go when computing a NGroup's value. TMAX pops up a menu containing this NGroup name and all its parents. You control how much of the NGroup value to display by selecting the first NGroup to be used in this NGroup's value. Note that this subitem doesn't actually change the NGroup value; it only determines how much of this value to display. Since top level NGroups only have a single value, you cannot abbreviate them. The subitems listed here are numbered with the NGroup "Sect Cntr" which as been abbreviated such that it only prints its own value without any of its parents values.

[Sect Cntr] `Starting Value`

This is the starting value of the NGroup. TMAX prompts for the new value in the prompt window rather than using a menu. The default `Starting Value` is 1.

[Sect Cntr] `Table-Of-Contents`

This is a flag that says whether or not this NGroup will be included in the Table-Of-Contents file should you decide to create one. The default is to include all NGroups in the Table-Of-Contents file.

[Sect Cntr] `Manual Index`

This is a flag that says whether or not this NGroup should be included in the manual-style index. Note that this item is only offered if the ***Manual Index*** (see section <Section/Value>) toggle switch is on. The default is to not include any NGroups in the manual-style index.

[Section] NGroup Delimiters

The `Delimiter Before/After` construct allows TMAX to support any numbering format. Since the default `Delimiter Before` is a null string and the default `Delimiter After` is a period, the default number format is of the form "1.", "1.2.", "1.2.3." etc. You may have noticed that this document uses this default form for the `Chapter` numbers but not for the `Section` numbers. To see how we did this just middle button `Chapter` in the NGroup graph above and select the `Show Format` item from the pop up menu. Then do the same for `Section`. The field following `"Display="` consists of the `Delimiter Before`, the `Display Type`, and the `Delimiter After` in that order.

There is one small anomaly with delimiters, namely the case where one NGroup has a `Delimiter After` (e.g. `Chapter`) and the following NGroup has a `Delimiter Before` (e.g. `Section`). If the `Delimiter Before` is not a null string then it will *always* override the `Delimiter After` in the preceding NGroup regardless of what the `Delimiter After` is. For example, if the `Delimiter After` for `Chapter` was a colon and the `Delimiter Before` for `Section` was a dash then chapter numbers would look like "1:" and section numbers would look like "1-2.".

[Section] Inserting Number Groups

You create NGroup members by buttoning **New NGroup**. To insert an NGroup member into your document, simply point the mouse to the appropriate name in the Number Group menu graph and ***left*** button it. The NGroup member name enclosed in square brackets (e.g. `[Chapter]`) will be inserted at the current cursor position in whatever font you have specified for this member. All NGroup markers that have not been updated are enclosed in square brackets to distinguish them from normal text. TMAX will warn you if you insert NGroups out of order. For example, in this document you should not insert "`Section`" until you insert "`Chapter`" because the value of `Section` depends on the value of `Chapter`.

[Section] Customizing inserted Number Groups

Although the font/format of the prototype NGroup determines the font/format of the inserted NGroup, you can change the font and certain parts of the format after the NGroup is inserted. Any change you make is *local* to that particular NGroup; it does not affect any other NGroups. If you middle button an inserted NGroup, a menu will pop up allowing you to define a Tag and show/change the font and format of the selected NGroup. If the NGroup already has a Tag, this menu will give you the option of changing, deleting, or displaying the Tag. The `Show Font` and `Change Font` items work exactly the same as those for the the prototype NGroups but the `Show Format` and `Change Format` are slightly different. The `Starting Value` and the `Table-of-Contents` and `Manual Index` flags can only be set for the prototype NGroup. You cannot change these for a particular inserted NGroup.

There are six subitems for `Change Format`. You can change any part of the format on an inserted NGroup by selecting one of these subitems. If you button `Change Format` instead, TMAX will prompt you for all six values. Four of the values use pop-up menus and the other two request input in the prompt window. The six parts of an inserted NGroup's format are:

[Sect Cntr] `Delimiter Before`

This works exactly the same as the `Delimiter Before` subitem for the prototype NGroups except the change doesn't affect any other NGroups.

[Sect Cntr] `Display Type`

This works exactly the same as the `Display Type` subitem for the prototype NGroups except the change doesn't affect any other NGroups.

[Sect Cntr] `Delimiter After`

This works exactly the same as the `Delimiter After` subitem for the prototype NGroups except the change doesn't affect any other NGroups.

[Sect Cntr] `Abbreviate Level`

This works exactly the same as the `Abbreviate Level` subitem for the prototype NGroups except the change doesn't affect any other NGroups.

[Sect Cntr] `Text Before`

This allows you to add, change, or delete the text string preceding the NGroup regardless of the setting of the ***Text Before*** toggle switch. TMAX will prompt you for the new string in the prompt window. A bare carriage return deletes the text before the NGroup. If you do specify a string, remember that TMAX will always append a space to the end of the string. Unlike inserting a NGroup with ***Text Before*** on, there is no default string.

[Sect Cntr] `Text After`

This allows you to add, change, or delete the text string following the NGroup regardless of the setting of the ***Text After*** toggle switch. TMAX will prompt you for the new string in the prompt window. A bare carriage return deletes the text after the NGroup. If you do specify a string, remember that TMAX will always append a space to the beginning of the string.

[Section] Pruning the NGroup graph

As mentioned before, the entire NGroup graph is saved/restored over a **Put**/**Get** even if some of the NGroup members aren't used in the document. We have described how to add members to this graph but not how to delete them. TMAX has two ways to delete unused NGroup members from the graph.

The first way is to `Copy` text (including the inserted NGroups) from one TEdit window to another. TMAX will only copy enough of the NGroup graph to handle the NGroups in the copied text. For example, you can remove every unused NGroup member from the graph by copying the entire document to another TEdit window.

The other way is to set the global variable `TMAX.PRUNE.NGRAPH` to `T`, open a TEdit window, and then **Get** the document. On a **Put** TMAX writes out the NGroup graph data structure. On a **Get**, TMAX checks this flag and if it is true, it creates the NGroup graph from the NGroups that appear in the document rather than from the NGroup graph data structure. There is a potential problem with using this feature. TMAX creates the NGroup graph from the first occurrence of each NGroup in the document. If the first occurrence of a particular NGroup has been modified (see section <Section/Value>) then TMAX will use this modified format as the new prototype format. There are no problems if the first occurrence of each NGroup has not been modified.

Chapter [Chapter] Contents File

One of the benefits of using NGroups is the Table-Of-Contents (TOC). By default all NGroups are included in the TOC along with their corresponding ***Text Before*** and/or ***Text After*** strings if any. If you don't want certain NGroup members included in the TOC, you can specify this by changing the NGroup's prototype format. Although this document has several NGroups, we have decided to include only the `Chapter` and `Section` NGroups in the TOC. The page numbers in the TOC are always printed in the Gacha 10 Standard font.

[Section] Create TOC

Buttoning this item creates a TEdit Table-Of-Contents file. You must first specify the name of the TOC file via the **TOC Filename:** item. Each line in the TOC consists of the ***Text Before*** string (if any), the NGroup number, the ***Text After*** string (in any), a dotted leader, and the page number on which this NGroup appears. Note that the TOC file itself does not contain any TMAX features. It is a simple TEdit text file.

[Section] View TOC

Buttoning this item first creates the TOC file (via **Create TOC**) and then prompts you for a TEdit window where TMAX displays the TOC file it just created.

[Section] TOC Filename:

This item allows you to specify the name of the TOC file. When this item is buttoned, the cursor will appear just to the right of the colon. You then type the name of the TOC file and terminate it with a carriage return. You can edit this filename string anytime. If you have already terminated the string, just button the item again. The editing features available here are the same as those available in the EXEC window. If you **Get** a TEdit/TMAX file, the **TOC Filename:** defaults to that file with a **.TOC** extension.

NOTE... You must hardcopy your document before creating the TOC file. The reason is TMAX needs the page numbers for the TOC file but TEdit only knows the page numbers while the document is being hardcopied. During the hardcopy process, TMAX saves the page number for each NGroup member. If you create the TOC without first hardcopying your TEdit document, the page numbers in the TOC will be `NIL`.

Chapter [Chapter] Indices

TMAX allows you to insert index requests into your document and create a sorted file of these indices including the page number each index appears on. There are two types of index requests; simple and extended. The format of the index marker depends on the type of index request. But, regardless of the type of index, it is important to note that the index markers are only displayed in the TEdit window. If you hardcopy the TEdit document, you will ***not*** see these index markers. TMAX always encloses index requests in curly braces (i.e. { and }) to distinguish them from normal text. **Update** and ***Set AutoUpdate*** have no effect on index requests.

[Section] Index

When this item is buttoned TMAX will prompt you for the index "Key". It will then insert the marker "`{Index key}`" into your document at the current cursor position. TMAX uses these keys to sort the indices and the sorting is case independent. The key is also printed in the Index file along with the page number(s) it appears on. Currently all simple indices are printed in the Gacha 10 Standard font. For example, we have indexed the phrase "Indexing requests" in the following parentheses (). Of course you will not see this index marker if you hardcopied this document. You can change the index key by middle buttoning the index marker. TMAX will bring up a one item menu (`Change Index`). If you button this item, TMAX will prompt you for the new index key.

[Section] Extended Index

This is a fancy form of indexing. When this item is buttoned TMAX first prompts you for the key to sort on. TMAX then prompts you for the "Entry" and its font. This is what is printed in the Index file instead of the key. If you do not specify an entry, it defaults to whatever the key is and the font defaults to Gacha 10 Standard. Finally, TMAX prompts you for the index page number option. There are three options; print the normal page number, print a fixed page number that you supply, or don't print any page number at all. TMAX then inserts the index marker "`{Index Key=key, Entry=entry, Option}`" into your document. The option is "Yes" if the page number is to be included in the index file, "No" if the page number is not included, or the numeric value if a fixed number is to be used in the index file. For example, we have (extended) indexed the word "Spies" but we want "Boris & Natasha" printed instead in the Helvetica 12 Italic font. The extended index is enclosed in the following parentheses (). Of course you will not see this extended index marker if you hardcopied this document. You can change any of the fields in an extended index request by middle buttoning the index marker. TMAX will bring up a one item menu (`Change Extended Index`). If you button this item, TMAX will prompt you for the new index key, entry, font, and number values.

[Section] Known Indices

Buttoning this item brings up a menu of all the indices and extended indices specified so far in alphabetical order. You can insert another **Index** or **Extended Index** request by simply buttoning the appropriate item in this pop-up menu. This makes it trivial to index the same items throughout a document. Indices are simple items but extended indices have subitems because several extended indices can have same key but completely different entries, fonts, and/or page number options. To insert an extended index, you must button the appropriate subitem; buttoning the extended index item has no effect. The extended index subitem shows the entry, font, and page number option. If you button **Known Indices** above, you will see the two indices used here as examples.

[Section] Manual Index

When this toggle switch is on, the page numbers in the index file are printed in "manual format". By manual format we mean something like "III:25.7" for chapter 3, section 25, page 7 (assuming the chapter's format has been changed to print Roman numerals following by a colon). You specify which NGroup members are included in the manual index page number. To include a NGroup member in the manual format page number, first make sure this toggle switch is on. Then change the format of the NGroup members you want included. The code to change the NGroup format checks the ***Manual Index*** switch setting. If the switch is on, the code asks if you want the selected NGroup member included in the manual index. You can only include the members of one major branch of the NGroup graph in the manual index page numbers. Using this document's NGroup graph as an example, you can include either or both members in the `Chapter-Section` branch ***or*** the single member in the `NG Format` branch but ***not*** both. TMAX will only allow you to include an NGroup member in the manual index if there are no other members included yet or the included members are in the same branch as the new member. The reason is there is no correlation between the numbers in disjoint branches in the NGroup graph. If you don't specify manual indexing, TMAX defaults to "book format" indexing. With book format the index file page references are just the page numbers themselves.

Chapter [Chapter] Indices File

These commands allow you to write out an index file sorted by the index keys. As we mentioned before, the sorting is case independent. The page numbers (for both "book" and "manual" style) are always printed in the Gacha 10 Standard font.

[Section] Create Index

Buttoning this item creates a TEdit sorted index file. You must first specify the name of the index file via the **Index Filename:** item. For simple indices each line in the index file consists of the index key followed by the page number(s) on which this index key appears. Extended indices are treated a little differently. Each extended index is printed on a separate line. Note that the index file does not contain any TMAX features. It is a simple TEdit text file.

[Section] View Index

Buttoning this item first creates the index file (via **Create Index**) and then prompts you for a TEdit window where TMAX displays the index file it just created.

[Section] Index Filename:

This item allows you to specify the name of the index file. When this item is buttoned, the cursor will appear just to the right of this item. You then type the name of the index file and terminate it with a carriage return. You can edit this filename string anytime. If you have already terminated the string, just button the item again. The editing features available here are the same as those available in the EXEC window. If you **Get** a TEdit/TMAX file, the **Index Filename:** defaults to that file with a **.INDEX** extension.

NOTE... You must hardcopy your document before creating the index file. The reason is TMAX needs the page numbers for the index file but TEdit only knows the page numbers while the document is being hardcopied. During the hardcopy process, TMAX saves the page number(s) on which each index appears. If you create an index file without first hardcopying your TEdit document, the page numbers in this index file will be `NIL`.

Chapter [Chapter] Specifying a Font

TMAX uses the same mechanism whenever it prompts you for a font. To specify a font you must select a value from each of three different menus. The first pop-up menu is used to specify the font family. The font families are Classic, Gacha, Helvetica, Modern, and Times Roman. After you select the family, the font size menu pops up. This menu contains the sizes 6, 8, 10, 12, 14, 18, 24, and 36. After you select the size, the font face menu pops up. The font faces are Standard, Italic, Bold, and Bold Italic.

The default for any value not specified (i.e. the mouse is buttoned outside the pop-up menu) is whatever it was before. The default font always starts out as Gacha 10 Standard. Therefore, if you select a new family (say Helvetica) and button the mouse outside the font size and face menus, the new font would be Helvetica 10 Standard. Now if you were to button the mouse outside the font family and size menus and select a new face (say Bold), then the new font would be Helvetica 10 Bold.

Some TMAX functions (like **Set Style**) require you to select values from three different menus. Other functions (like changing a NGroup font) allow you to change one of the three fields directly without changing the other two. This is exactly equivalent to selecting a value from one menu and buttoning the mouse outside the other two menus.

Chapter [Chapter] Random thoughts and hints

- You can `Copy` any of the TMAX features. from one TEdit window to another and TMAX will automatically set up the internal data structures necessary to support that feature in the destination window. For example, if you `Copy` a NGroup marker, TMAX will automatically set up the NGroup graph. Note that it only sets up enough to support what was copied; TMAX does not set up the entire NGroup graph.

- TMAX does not support `Move`. If you want to move any TMAX features, you will have to `Copy` and then delete the features.

- If you just want to select a TMAX feature, you should do so with the left mouse button. The right button will select the feature as well but it also causes TMAX to pop up a menu. Every TMAX feature pops up a menu if it's middle buttoned.

- Regarding forward page References and doing a "dummy" hardcopy, we do the dummy hardcopy to a file whose device is `{NODIRCORE}` and whose extension corresponds to the eventual hardcopy device (e.g. `{NODIRCORE}X.INTERPRESS`). `NODIRCORE` is the bit bucket and the extension causes TEdit to load the appropriate fonts so the real hardcopy will go a little faster.

- Remember that changing the font/format of a prototype NGroup will also change every occurrence of that NGroup in the document. Therefore you should settle on the font/format of the prototype NGroups before you start customizing any inserted NGroups (see section <Section/Value>).

- Occassionally TEdit (not TMAX) has problems displaying some TMAX features. For example, if you change the prototype NGroup font then every occurrence of that NGroup also changes. Sometimes it looks like only the first occurrence has changed when in fact they all have. If you `Redisplay` the window, it will look like it should. It is a good idea to `Redisplay` the TEdit window whenever you think it doesn't look right.

TMAX Introduction

TMAX (Tedit Macros And eXtensions) is a package that extends the capabilities of TEdit by providing a menu of commands that allow the user to do such things as indexing, writing a sorted index file, arbitrary numbering, creating a list of notes, referencing numbered objects by their numeric value, and writing a table-of-contents file. There is also an extensive help file describing all the features of TMAX. Rather than include examples of how to use these features, the help file was written using these features and is both an example and a description of using TMAX.

Before you can read the help file, you must first load the TMAX.DCOM file from the {CSLI}<LISP.KOTO> directory. Then copy {CSLI}<LISP.KOTO>TMAX.TEDIT to your local disk. Open up a TEdit window and do a TEdit Get on {DSK}TMAX.TEDIT. The reason for copying the file to the local disk is that, due to problems with the leaf server on csli, TMAX runs a lot smoother when the source files are on non-leaf hosts.

When you load the TMAX.TEDIT file, you will see the TMAX menu appended to the top of your TEdit window. You invoke TMAX commands by buttoning items in this menu. If you would like a hardcopy of the help file, first button Update and then Insert Endnotes (both in the TMAX menu). Then point the mouse at the black title bar at the top of the TEdit window and select Hardcopy from the right button menu.

TMENU

By: Mark Stefik, Daniel G. Bobrow, and Chris Tong (Stefik.pa@Xerox.com)

The TMENU package provides the following features:

TMenus.

These are interactive menus intended for reducing the amount of typing to an Interlisp-D program. When an item is selected in a menu, an expression is inserted into the TTY input buffer. It can be used to simplify the entry of common LISP commands or long names.

Windowshades.

Windowshades are a modification that can be made to any window in order to conserve screen space. They make a window "roll up" when not in use, leaving behind only the title. When the mouse is clicked inside the remaining bar, the window unrolls for interaction, and rolls up again at completion. Windowshades can be used with TMenus.

Msgs to the Prompt window.

Two functions are provided for clearing and printing to the Interlisp-D prompt window. These functions take an arbitrary number of arguments and preserve the black background shade of the window.

INTERACTION WITH TMENUS

TMenus are placed on the display under program control using the functions described below.Once a menu is on the screen, items may be selected using the LEFT mouse button. This causes some text to be inserted into the teletype buffer. In the default case, this text is just the item in the window, but it can alternatively be the result of evaluating an expression. In the default case, the text is followed by a blank space in the buffer, but it can be followed alternatively by an arbitrary string (such as the empty string or a carriage return). Non-default cases are controlled by arguments to the TMenu function.The MIDDLE mouse button is used for user-interactions that change the menu. When the middle button is depressed inside a menu, another pop-up menu (a meta-menu!) appears that provides several options for changing the menu, such as adding or deleting items. One of the options is to recompute the set of items by evaluating an expression associated with the menu.The RIGHT mouse button is used for the usual window commands. These commands work inthe standard way except that when a TMenu is reshaped, internal functions are invoked to adjust the configuration of rows and columns in order to create a menu that is visually appealing.

MAIN FUNCTIONS

TMenu (*itemExpr title displaySpec windowShadeFlg buttonFndefaultTrailerString*)

[Function]

TMenu is the function for creating a menu in a window on the display. Its arguments are as follows:

itemExpr an expression for computing the list of items that is saved with the menu. In the usual case,itemExpr is a list, whose items are either atoms or lists of the form:(displayThis evalThis comment trailerString)where displayThis is the form displayed in the menu, evalThis is the form evaluated to createthe text for the input buffer, comment is displayed in the prompt window if the LEFT button is held over an item for an extended period, and trailerString is the string inserted after anexpression in the input buffer. Most of these fields are optional. The default value for evalThisis the entry in displayThis. The default trailerString can be specified for a TMenu in thedefaultTrailerString argument above. Otherwise it is a space if the expression is an atom, and the empty string otherwise. This definition of fields for menu items is compatible with the usual set for the Interlisp-D menu package, with the addition of the trailerString field.

Special cases:

If itemExpr is an atom, it must be the name of a global variable to be evaluated to yield the list of the form described above (e.g., MYFNS). If itemExpr is a list and the first element of the list has a functional definition, then that function is evaluated to yield the list.This is intended for cases where the list is to be computed.

title The title of the menu. This title is used as the title of the window containing the menu.

displaySpec This argument has several possible interpretations that control the display of the menu. IfdisplaySpec is a region, then the window for the menu is placed in that region. If displaySpec is a number, that number is used as the number of columns in the menu display and a minimum size window is allocated for displaying the entire menu. The user is prompted with a ghost box to place the menu on the display. If displaySpec is T, then the number of columns is computed by TMenu assuming a maximum of 15 rows per column and the user is prompted for placement as before. If displaySpec is NIL, then the user is prompted to place a bounding box for the menu and TMenu tries to compute an arrangement of rows and columns that is visually pleasing.

windowShadeFlg If T, the window containing the menu is augmented with a window shade so that the menu"rolls up" if not in use. If windowshadeFlg is NIL, the menu is placed in a window on the screen.

buttonFn Optional argument that allows the caller to specify his own function for handling the LEFT andMIDDLE buttons.

defaultTrailerString Optional argument that allows the caller to specify the default string to follow each item printed. Can be overridden for specific items by the trailerString argument in the itemExpr.

EXAMPLE

The following expression:(TMenu ’MYFNS "Common Fns" T) would create a window titled "CommonFns" and display the list of functions in the window. The window would contain columns of up to 15 functions each, and would be placed on the screenunder user control. If the user later used the MIDDLE button to add or delete items from the menu, then the list in the variable MYFNS would be updated as the menu is updated.

MakeFileMenus (*fileName*) [Function]

MakeFileMenus is the function for creating a set of menus for the functions and variables in afile. A window is created for each menu. The menus are placed under user control and are all given window shades. The argument fileName is the name of a file.

Example

If the file is MYFILE, then the fns on MYFILEFNS and the vars on MYFILEVARS would be displayed in menus. MakeFileMenus would look for file commands of the form (FNS * FnsLst)and (VARS * VarsLst) on the command.

CloseFileMenus (*fileName*) [Function]

Closes all of the TMenu windows associated with the given file.

Window ShadesMakeWindowShade (*window*) [Function]

Modifies the given window to provide a window shade. If the window argument is NIL, then the window is selected which is under the cursor. If the window argument is T, it waits for the CTRL key to be depressed, and then selects the window under the cursor.Prompt Window FunctionsPROMPT (arg1 arg2 arg3 ...)Prints an arbitrary number of arguments to the prompt window, after first clearing the window.A call with no arguments simply clears the window.

CPROMPT (*arg1 arg2 arg3* ...) [Function]

Same as prompt except the arguments are printed centered in the window.

Trajectory-Follower

By: D. Austin Henderson, Jr. (AHenderson.pa@Xerox.com)

INTRODUCTION

Trajectory-Follower provides a function which causes a "snake" to crawl along a trajectory. Comments on both interface and functionality are welcomed.

FUNCTIONS

(TRAJECTORY.FOLLOW *KNOTS CLOSED N DELAY BITMAP WINDOW*) [Function]

The trajectory is specified by *KNOTS* (a set of knots) and *CLOSED* (a flag indicating whether it is an open or closed curve). *N* is the length of the snake in points along the curve. *DELAY* is the time (in milliseconds) between each move along the curve; *DELAY* = 0 or NIL means go as fast as you can. *BITMAP* is the brush to be used at each point in creating the snake. *WINDOW* is the window in whose coordinate system the knots are given and in which the snake is to be drawn; if NIL, then the SCREEN bitmap is used. The snake is moved by INVERTing the bitmap at the points along the curve, and then INVERTing the bitmap back out again.

Examples

A demonstration function is also provided with the module:

(TRAJECTORY.FOLLOWER.TEST) [Function]

Interacts with the user through prompting in the promptwindow to gather up arguments for TRAJECTORY.FOLLOW and then carries it out. Closed curves are snaked around repeatedly until the left shift key is found depressed when it reaches the curve’s starting point.

Internal Functions

The internal functions used by this module are also available for use. They are:

(TRAJECTORY.FOLLOWER.SETUP *WINDOW N DELAY BITMAP*) [Function]

Initializes drawing variables.

(TRAJECTORY.FOLLOWER.POINT *X Y WINDOW*) [Function]

Defines the next point on the curve. Note that the argument structure of this function is appropriate for use as a BRUSH with the curve drawing functions DRAWCURVE, DRAWCIRCLE, and DRAWELLIPSE. (For an example, see the demonstration function TRAJECTORY.FOLLOWER.TEST)

(TRAJECTORY.FOLLOWER.WRAPUP) [Function]

Finishes the job after all the points have been defined.

Sender: SenderNameTooLong%SUMEX-AIM.STANFORD:EDU:Xerox
Date: 5 Jan 88 18:39:54 PST (Tuesday)
Subject: Interlisp to Commonlisp conversion package
From: darrelj%sm.unisys:COM
To: info-1100%sumex-aim.stanford
cc: darrelj%sm.unisys:COM

Return-Path: <@SUMEX-AIM.Stanford.EDU:sdcrdcf!sm.unisys.com!darrelj@PRC.Unisys.COM>
Redistributed: Info-1100-Redistribute^.X
Received: from SUMEX-AIM.Stanford.EDU by Xerox.COM ; 05 JAN 88 18:38:18 PST
Received: from burdvax.PRC.Unisys.COM (PRC-GW.PRC.UNISYS.COM) by SUMEX-AIM.Stanford.EDU with TCP; Tue, 5 Jan 88 17:44:10 PST
Received: by burdvax.PRC.Unisys.COM (burdvax) [5.54/1.0]   id AA15346; Tue, 5 Jan 88 20:44:28 EST
Received: by rdcf.sm.unisys.com (sdcrdcf) (5.51/Domain/jpb/2.7)   id AA28679; Tue, 5 Jan 88 16:35:37 PST
Message-Id: <8801060035.AA28679@rdcf.sm.unisys.com>
Received: from XAVIER by sdcrdcf with PUP; Tue, 5 Jan 88 16:35 PST
Original-Date: 5 Jan 88 16:34 PST (Tuesday)

Interlisp-D into Xerox Commonlisp, we developed a collection of tools to automate the conversion as much as possible.
These have been placed in {parcvax.xerox.com}/lisp/exchange.
While we at Unisys have reasonable confidence in these tools, they are being made available with no promises of accuracy, completeness or support (though we would appreciate feedback).

The tools run in Xerox Lyric Common Lisp. The following files are parts of it:

TRANSOR -- A slightly modified version of Transor, to fix a few Lyric-related problems and provide the ability to emit a DEFINE-FILE-INFO expression and to use the value of TRANSOUTREADTABLE as the output readtable. We used TRANSOR because we were familiar with it, and it handles a lot of details needed to safely and surely traverse the code to be translated. The biggest impediment to adding to the transforms is that they are specified as teletype editor commands, and only old-time Interlispers have much experience with those.
TSET -- The same version dating back to 1979. This is the part of transor used for interactively developing and testing translation rules.
TRANSOR.LCOM -- contains the compilation of BOTH the above files.
TO-COMMONLISP.XFORMS -- translation rules for 428 functions, 98 remarks and 4 auxiliary functions. It covers are large portion of Interlisp, including most Clisp constructs, and specifically handles any function with the same name in both Interlisp and Commonlisp, so that holes in a translation should result in calls to undefined functions. In many cases, nice transformations are used for easy cases, and ugly ones only for hard cases. This file sets TRANSOUTREADTABLE to be a copy of the XCL readtable which is case SENSITIVE, MYLOAD below reads it case INsensitive, so the resulting file will ultimately lose most case distinctions on reloading into Xerox Lisp (or other common lisps). This was a much debated point internally, but this seemed the best of three bad possibilities (e.g. print one of
Cased AS ORIGinal   which becomes CASED AS ORIGINAL on load,
|Cased| AS |ORIGinal|, or
CASED AS ORIGINAL

INTERLISP-COMMONLISP.TEDIT -- A document describing the transformations and formacro.

LOADTRAN -- contains a few functions which prevent many breaks on loading the translated file. The function MYLOAD is intended to load

a translated file.
LOADTRAN.DFASL -- compiled version

FORMACRO and FORMACRO.DFASL -- Still another portable iteration macro for commonlisp. Its main claims are almost 100% compatibility with the semantics of the Interlisp-Clisp FOR (especially when used the the XFORMS which fix a few incompatibilities); and user extensibility (unfortunately not compatible with IL:I.S.OPR). Embedded keywords (e.g. IN, COLLECT) may be in any package.

COMMON-MAKE and COMMON-MAKE.LCOM -- still another version of code to generate a more "common" source file. It handles more filepkg command types than most. Also, when used with COMMENTHACKS will successfully print ALL comments in semicolon format. Call IL:COMMON-MAKEFILE(file). It checks the MAKEFILE-ENVIRONMENT property to select a package and base.

COMMENTHACKS and COMMENTHACKS.LCOM -- patches to the prettyprinter and to the DEFUN editor. The prettyprinter patches will print Interlisp (* --) comments as semicolon comments when *PRINT-SEMICOLON-COMMENTS* is 'IL:ALL.
This file also redefines the ED method for DEFUNs so that the initials and date of editing get updated for DEFUNs just as Interlisp has always done for FNS.

Because of the way things developed, these tools are not as fully integrated as they could have been. If we were doing it over, the TRANSOR step could have more carefully coordinated the new COMS so that COMMON-MAKE would be able to do the right thing. As it stands, the COMS generally have to be edited to change FNS to FUNCTIONS, etc, but you tend to need a few iterations of editing things before the compiler is completely happy anyway.

The steps needed to do translations are roughly as follows:

```
(LOAD 'TRANSOR.LCOM)
(LOAD 'TO-COMMONLISP.XFORMS)
(SETQ FIXSPELLDEFAULT 'N)  ;; Otherwise DWIM gets too clever
(SETQ XlatedRecords NIL)   ;; This is currently set to records
specific to the system we translated.
```

TRANSOR files containing record declarations. The records MUST be translated before any code containing create/fetch/replace since the translation depends on the type of records. Also, the record declarations should be LOADED. In a large translation effort, save a file containing all needed declarations and the value of XlatedRecords computed by translating them.

```
(TRANSOR 'file1) ...  ;; results in file1.TRAN and file1.LSTRAN, see
TRANSOR documentation.
```

To load translated files into a fresh xerox lisp system:

```
>From an XCL exec:
(IL:SETPROPLIST '*COMMENT* (IL:GETPROPLIST 'IL:*))
(IL:PUTASSOC '*COMMENT* 'IL:* IL:PRETTYEQUIVLST)
(LOAD 'LOADTRAN)
(SETQ IL:*DEFUALT-MAKEFILE-ENVIRONMENT* '(:READTABLE "XCL" :PACKAGE ???))
(SETQ IL:CMLRDTBL (IL:FIND-READTABLE "XCL"))
(LOAD 'FORMACRO.DFASL) if used interlisp for's
  ;; may need to import USER:FOR depending on packages you've set up.
(MYLOAD 'translated-records)
(MYLOAD 'file1.tran) ...
```

A little work with ED and FILES? and you should be able to save a commonlisp version of your files (well, OK, a lot of work).

Suggestions and questions to one of:
darrelj@RDCF.SM.UNISYS.COM or darrel@CAM.UNISYS.COM,
or fritzson@bigburd.prc.UNISYS.COM

----- End Forwarded Messages -----

TRICKLE

By: Nick Briggs (Briggs.pa@Xerox.com)

Uses: PROMPTREMINDERS

This document last edited on October 12, 1987

INTRODUCTION

Trickle provides a very simple cover for COPYFILES to do periodic (every 24 hours) updating of one directory from another, with processing of the log files generated by COPYFILES to mail a note to some designated person indicating what COPYFILES did.

USE

There is only one function of interest to the user:

(Trickle *Source Destination RootLogfileName MailAddress ScheduleAnotherOne DontReplaceOldVersions*) [Function]

Source and *Destination* should be patterns acceptable to COPYFILES. *RootLogfileName* should be a host, directory, and partial file name to which Trickle will append the date in the form yymmdd, and the extension .CopyLog. On completion of the copy Trickle will mail a message to *MailAddress* if it is non-NIL. If *ScheduleAnotherOne* is T then another Trickle will be scheduled (randomly) between 1 am and 5:59 am of the next day, alternatively, if *ScheduleAnotherOne* is a time that would be acceptable to IDATE (Trickle will prepend the actual date, just give the time) then another Trickle will be scheduled at exactly that time. *DontReplaceOldVersions* signals that Trickle should not use the COPYFILES option REPLACE, use of which causes problems with NS file servers (at least in Koto).

Example

To update the directory {cf}<lispusers>koto>* from {eris}<lispusers>koto>* storing the log files starting with {core}eluk-870512.copylog, mailing notification to Briggs.pa, scheduling this to run every night, and using COPYFILES' REPLACE option one would execute:

```
(SETREMINDER NIL NIL
             '(Trickle '{eris}<lispusers>koto>* '{cf}<lispusers>koto>*
                     "{core}eluk-" "Briggs.pa" T)
             "12-May-87 03:00")
```

Two versions of the log file will be created; version 1 with the complete log output of COPYFILES, and version 2, with all the "skipped" files removed. It is this version that is mailed to the designated recipient.

The mail messages that are sent out indicate whether there were any files processed: the subject line will include the string "(Empty)" if no files were Trickled, and the string "(Error?)" if there were no files in the source directory (may not be an error, but may be worth investigating)

TURBO-WINDOWS

By: Andrew J. Cameron, III (Cameron.pa@Xerox.com or cameron@cs.wisc.edu)
New Owner: Atty Mullins (Mullins.pa@Xerox.com)

Uses: WDHACKS (LispUsers) [optional]

This document last edited on Sept. 8, 1988.

INTRODUCTION

Turbo-Windows does not have anything to do with speeding up primitive window operations, but rather it helps speed up your use and manipulation of windows by providing most of the right button menu functions via shift keychords. In this way one can Move, Shape, Copy, Shrink, Close, etc., a window without having to wait for the right button menu to appear and then select from it.

Also, when providing the INITIAL shape of a window, pressing the middle button yields a large default size suitable for TEdit, etc. (Recall that using the middle button during a RESIZING operation allows you to keep roughly the original window shape and then move the corner nearest the cursor when the middle button was pressed.)

One can bring up a brief cribsheet for all the TurboWindow keychord commands by holding down the HELP key and RIGHT buttoning on the background (not in any window). This can also be produce by typing (TW.HELP) to an InterLisp EXEC.

OPERATION

Before discussing how to use this utility, a description of how the key-chords were chosen is in order. They are based loosely on the effect of the shift keys in TEdit. Recall that in TEdit, pressing and holding the Shift key Copies whatever is selected. Also, pressing and holding the Control key (sometimes labeled PROPS or EDIT) Deletes whatever is selected. Pressing both Shift and Control performs both a copy and a delete, which ends up Moving the selected item. The only additional piece of information that you need to know is that the Meta key (sometimes labeled KEYBOARD or ALT) modifies an operation or in some way makes it different. With this general interpretation, most of the key-chords are rather easy to remember.

If the following keys are chorded (held down together) while the right mouse button is pressed in the region of a window which would normally bring up the right-button menu (by convention, at least the title bar should provide the right button menu), the listed operation will be invoked without actually bringing up the right button menu.

SHIFT (using the LEFT SHIFT key or CAPS LOCK key)

Makes a copy of a window by snapping it.

CONTROL

Closes (deletes) a window. (Since this is a destructive operation, a small safeguard is built into this operation. If one holds the CONTROL key and depresses the right mouse button and continues to hold them, the window to be operated on (closed) will blink. If this is not the window you want to close you can cancel the Turbo-Close by either moving outside the window (or by releasing the CONTROL key before releasing the right mouse button). If you abort the Turbo-Close in this manner, the normal right button menu will appear. Clicking outside of the menu will make it go away. Sometimes unexpected things occur when trying to Turbo-Close windows with attached windows, e.g. FileBrowsers, but hopefully this safeguard is conservative enough to avoid inadvertent closing of the wrong window.) [Holding down CONTROL while Right Buttoning on the background activates Window Slamming, if the LispUsers utility WDHACKS is loaded.)

META

Shape (makes different) a window.

SHIFT-CONTROL

Moves a window. (Due to the design of the InterLisp window system, this operation works in a rather strange way. You press and hold both CONTROL and SHIFT and then press the right mouse button while in the appropriate part (title bar) of the window you want to move. You then need to release the right button to be able to actually move the window. In order to "drop" the window (here is the strange part) you need to press the LEFT (or middle) button. Pressing the right button merely allows you to move to a different corner of the shadow box.)

META-CONTROL

Shrinks ("deletes" in a different way) a window.

META-SHIFT

Redisplays (copies in a different way) a window.

META-SHIFT-CONTROL

Buries (moves in a different way) a window. [You might also think of this as pushing the window down to the bottom, as you are pressing down all three shift keys.]

RIGHT-SHIFT

Clears a window.

HELP

Pressing the HELP key while the cursor is in the background (or typing (TW.HELP) to an InterLisp EXEC) displays a cribsheet for the Turbo-Window KeyChords. Some addition capabilities not listed here are given on that cribsheet. The "OTHER" keychords which are marked with an asterisk (*) indicate that some side-effect (potentially quite harmful) might occur depending on where the TTY is when those alternate access methods are used. You are warned!

GETTING STARTED

[If any of the operations described below do not perform properly, it might be the case that your keys are not defined in the way that this utility expects. See INTERNALS below for more information.]

You might want to get familiar with Turbo-Windows by first bringing up the cribsheet by depressing the HELP key and right-buttoning on the background. Next, make a copy of the cribsheet by depressing SHIFT (the left shift key) and right-buttoning on the cribsheet. Drop the new copy of the cribsheet by releasing all keys and buttons and the pressing the left mouse button. [Note: The cribsheet is merely written to the TTY window, which happens to be sensitive to the right mouse button everywhere. Other windows may only be sensitive to the right button (for the purpose of bringing up the right button menu, in their title bar.] Now try moving the copied cribsheet by pressing both SHIFT and CONTROL

(PROPS or EDIT) and right-buttoning on the copy of the cribsheet. Again, release everything (well, just releasing the mouse button will do) and press the left mouse button to drop it. Press and hold both META (KEYBOARD) and CONTROL while right buttoning in the copy of the cribsheet to shrink it to an icon. Release and click the left mouse button to drop the icon. Reopen (expand) the icon by middle buttoning on it. Reshape the copy of the cribsheet by pressing META and right buttoning on the copy's window. Release and rubberband the new shape with the left mouse button. (Do you know what would happen if you used the middle button after releasing instead? Try it.) Assuming the copy of the cribsheet is overlapping another window and some part of the background (if not, Turbo-Move it so it is), press and hold all three (META, SHIFT, and CONTROL) and right button in the cribsheet copy's window to bury it. Right button (holding no other keys) in the partially exposed area of the now buried cribsheet copy to bring it back to the top. Finally, close the copied cribsheet window by pressing CONTROL while right buttoning in the copy's window. [O.K. which shift key combination hasn't been used yet? Consult the original cribsheet (or produce it again), if necessary. Give that combination a try in the original cribsheet's window. [Did you notice the message in the prompt window?] And don't forget to give the Right Shift key (Clears a window) a try as well. [Remember, the cribsheet can be brought back at any time using HELP-RightButton on the background.]) To see how to cancel a Turbo-Close, depress the CONTROL key and press AND HOLD the right mouse button while in the original cribsheet window. Notice that the window blinks. Before you release the right mouse button move the cursor outside the cribsheet's window and then release the right mouse button. The cribsheet's window is not closed because releasing outside the window that flashed cancels the Turbo-Close. The normal right button menu appears instead. Click outside it to get rid of it. Now, actually close the original cribsheet window. And with that, may I welcome you to the fast paced world of Turbo-Windows.

INTERNALS

The right button events are intercepted by a piece of advice placed on DOWINDOWCOM. The middle button sizing capability is provide by advice on \GETREGIONTRACKWITHBOX.

The window snapping Turbo-Window feature (LeftShift-RightMouseButton) is also added as a submenu to the normal right button menu provided by the window system.

A common problem is that the META key is not defined to be at the proper place (attached to the key named KEYBOARD). To remedy this, type:

```
(KEYACTION 'KEYBOARD '(METADOWN . METAUP))
```

to an InterLisp EXEC. The following should also be the case:

```
(KEYACTION 'EDIT '(CTRLDOWN . CTRLUP))
(KEYACTION 'LSHIFT '(1SHIFTDOWN . 1SHIFTUP))
(KEYACTION 'RSHIFT '(2SHIFTDOWN . 2SHIFTUP))
```

These can be verified by using, for example:

```
(KEYACTION 'EDIT)
```

TW.NO-FLASH-CLOSE [Variable]

Initially NIL, if set to T, windows will not flash to indicate there impending closure.

TW.DONT-GROW-SNAP-BORDER [Variable]

Initially NIL, if set to T, windows will be copied without a small border. The small border is quite handy in telling the original window from its Turbo-Snapped copy.

TW.SNAP-HERE [Variable]

Initially NIL, if set to T, windows will be copied directly on top of the window they are duplicating. Normally (when NIL) the user must position the copy.

GETREGIONDEFAULT [Variable]

This variable can be bound dynamically by an application to provide the region afforded by middle buttoning when prompted for an initial region of a window. It is initially set to roughly 7x9 inches, and is useful for TEdit windows, FileBrowsers, etc. [See the LispUsers utility RESIZE-FILEBROWSER for an even better way of dealing with FileBrowsers.]

- In order to edit/compile the source of this utility, the InterLisp Source file WINDOW must be loaded in order to provide the SCREEN record definition used by the window system internals. The loading of this source file occurs automatically when this utility's source file is loaded.
- This utility interacts poorly with other utilities that redefine any of the shift keys. TEDITKEY and PC-Emulation (among others) are dubious in this regard.

TWODGRAPHICS

By: Jan Pedersen (Pedersen.PA @ Xerox.com)

Uses: UNBOXEDOPS

TWODGRAPHICS implements viewports. A viewport is a subregion of a window (or image stream) within which graphics is clipped and a linear transformation from a world coordinate system to the window (or image stream) coordinates.

A given window (or image stream) may have any number of viewports defined and the viewports may be arbitrarily nested or overlapping. If a window is reshaped the subregions of all currently defined viewports are proportionately reshaped.

Viewports will operate in the context of any image stream, (Interpress printers, etc.), although not all DIG (Device independent graphics) primitives are supported.

(CREATEVIEWPORT *stream streamsubregion source*) [Function]

Creates a viewport on stream. Stream is the target stream. Streamsubregion is a region in stream coordinates that defines the extent of the viewport.

Source may be a REGION in world coordinates, in which case the world to stream linear transformation is set up to map left to left and bottom to bottom, etc., or a VIEWPORT, in which case the new viewport inherits its world to stream transformation.

Returns a VIEWPORT

(SETWORLDREGION *region viewport*) [Function]

(Re)sets the worldregion of viewport and recomputes the transformation.

(SETSTREAMSUBREGION *region viewport*) [Function]

(Re)sets the streamsubregion of viewport and recomputes the transformation.

Modified versions of selected DIG primitives are supplied to take advantage of the world to stream transformation.

(TWODGRAPHICS.BITBLT *source sourceleft sourcebottom destinationviewport destinationleft destinationbottom width height sourcetype operation texture clippingregion*) [Function]

World coordinates may be used where it makes sense. The destination must be a VIEWPORT. Destination left and bottom default to the viewport's stream subregion left and bottom. The clippingregion argument is always in destinationviewport world coordinates. The source may be a VIEWPORT, a BITMAP, or NIL in the case of texture patterns.

In the following, all coordinates must be world coordinates.

(TWODGRAPHICS.MOVETO *x y viewport*) [Function]

(TWODGRAPHICS.MOVETOPT *position viewport*) [Function]

Here position is a POSITION in world coordinates

(TWODGRAPHICS.RELMOVETO *dx dy viewport*) [Function]

(TWODGRAPHICS.RELMOVETOPT *dposition viewport*) [Function]

(TWODGRAPHICS.DRAWTO *x y width operation viewport color dashing*) [Function]

(TWODGRAPHICS.DRAWTOPT *position width operation viewport color dashing*) [Function]

(TWODGRAPHICS.RELDRAWTO *dx dy width operation viewport color dashing*) [Function]

(TWODGRAPHICS.RELDRAWTOPT *dposition width operation viewport color dashing*) [Function]

(TWODGRAPHICS.DRAWLINE *x1 y1 x2 y2 width operation viewport color dashing*) [Function]

(TWODGRAPHICS.DRAWBETWEEN *position1 position2 width operation viewport color dashing*) [Function]

(TWODGRAPHICS.DSPRESET *viewport*) [Function]

Does a ‘‘DSPRESET‘‘ on the VIEWPORT

(TWODGRAPHICS.DSPFILL *region texture operation viewport*) [Function]

region must be in world coordinates

The following function is an extension which may be of use to those who wish to produce analytic plots.

(TWODGRAPHICS.PLOTAT *position glyph viewport operation*) [Function]

Bitblts glyph to position with operation, with glyph centered at position.

Several functions provide access to the world to stream transformations.

(WORLDTOSTREAM *position viewport oldposition*) [Function]

Position is in world coordinates. Oldposition is smashed if provided.

Returns the corresponding position in stream coordinates.

(WORLDREGIONTOSTREAMREGION *region viewport*) [Function]

Region is in world coordinates

Returns the corresponding region in stream coordinates

(WORLDTOSTREAMX *x viewport*) [Macro]

Returns x in stream coordinates.

Uses unboxed floating point arithmetic

(WORLDTOSTREAMY *y viewport*) [Macro]

Returns y in stream coordinates

Uses unboxed floating point arithmetic.

(WORLDXLENGTH *dx viewport*) [Macro]

Returns the length dx in stream coordinates

Uses unboxed floating point arithmetic.

(WORLDYLENGTH *dy viewport*) [Macro]

Returns the length dy in stream coordinates.

Uses unboxed floating point arithmetic.

(STREAMTOWORLD *position viewport oldposition*) [Function]

Returns position in world coordinates.

(STREAMTOWORLDX *x viewport*) [Macro]

Returns x in world coordinates.

Uses unboxed floating point arithmetic.

(STREAMTOWORLDY *y viewport*) [Macro]

Returns y in world coordinates.

Uses unboxed floating point arithmetic.

(STREAMXLENGTH *dx viewport*) [Macro]

Returns dx in world coordinates.

Uses unboxed floating point arithmetic.

(STREAMYLENGTH *dy viewport*) [Macro]

Returns dy in world coordinates.

Uses unboxed floating point arithmetic.

For those who desire tighter control over the two-stage process, transform into stream coordinates, and then clip against the viewport, the following functions provide primitive clipping for line drawing and text output in any image stream.

(CLIPPED.BITBLT *clippingregion source sourceleft sourcebottom destination destinationleft destinationbottom width height sourcetype operation texture*) [Function]

As in BITBLT, although the operation is clipped against clippingregion in destination stream coordinates.

(CLIPPED.DRAWLINE *clippingregion x1 y1 x2 y2 width operation stream color dashing*) [Function]

As in DRAWLINE, although the operation is clipped against clippingregion in stream coordinates.

(CLIPPED.DRAWTO *clippingregion x y width operation stream color dashing*) [Function]

As in DRAWTO, although the operation is clipped against clippingregion in stream coordinates.

(CLIPPED.DRAWBETWEEN *clippingregion pt1 pt2 width operation stream color dashing*) [Function]

As in DRAWBETWEEN, although the operation is clipped against clippingregion in stream coordinates.

(CLIPPED.PLOTAT *clippingregion position glyph stream operation*) [Function]

BITBLT glyph to stream centered at position and clipped against clippingregion.

(CLIPPED.PRIN1 *clippingregion expr stream*) [Function]

PRIN1 expr on stream clipped against clippingregion.

UNBOXEDOPS

By: Jan Pedersen(Pedersen.PA @ Xerox.com] and Larry Masinter (Masinter.PA @ Xerox.com]

The module UNBOXEDOPS is intended to assist those interested in high-performance, scalar, floating-point arithmetic. The basic trick is to perform floating point arithmetic on the stack, utilizing special, unboxed, floating-point opcodes, an ugly but usually effective solution. This method of eliminating floating-point number boxes is likely to change, but in the interim a combination of compiler declarations and explicit evocations of unboxed operations, as described below, will allow the interested user to eliminate a high percentage of floating-point number boxes. This module and the methods described are "safe", i.e., the declarations won't cause your programs to crash, and if it works with the declarations it will also work without them.

Unboxed floating point tricks help out only 1108's with floating point hardware or 1186's with floating point microcode. Unfortunately, they may make performance even worse on 1108's without floating point hardware, although the performance degradation is probably not too severe.

There exist opcodes which perform floating point arithmetic on the stack (that is, on the bits of those numbers, rather than pointers to those bits). These opcodes are only emitted by the byte compiler if arithmetic occurs in an unboxed context. One example of an unboxed context is arithmetic on a record field defined to be of type FLOATP, another is arithmetic on a variable declared to be of TYPE FLOAT . However, the compiler will box across function boundaries and in a return context. Furthermore, there exist more unboxed opcodes than are used by the compiler (unboxed comparison springs to mind).

UNBOXEDOPS defines macros/functions so that these additional opcodes may be exploited in an unboxed context. These macros/functions include:

UFABS, UFEQP, UFGEQ, UFGREATERP, UFIX, UFLEQ, UFLESSP, UFMAX, UFMIN, UFMINUS, and UFREMAINDER,

which behave identically to there non-U namesakes, except that the operations are done on the stack without generating floating point boxes.

For those unfamiliar with unboxed compiler declarations a short description follows:

Using (DECLARE (TYPE FLOATING x y z)) to reduce number boxes

Consider the silly function:

```
(DEFINEQ (FIE  (N)
         (bind (SETQ X 0.0) (SETQ Y 2.0) for I from 1 to N
                  do (SETQ X (FPLUS X (FTIMES Y Y)))
                  finally (RETURN X))))

(TIMEALL (FIE 100))
```

returns a CPU time of .025 and reports 200 FLOATP boxes produced. Now, consider

```
(DEFINEQ (FOO  (N)
        (bind (SETQ X 0.0) (SETQ Y 2.0) for I from 1 to N
                declare (TYPE FLOAT X Y)
                do (SETQ X (FPLUS X (FTIMES Y Y)))
                finally (RETURN X))))
 (TIMEALL (FOO 100))
```

returns a CPU time of .003 seconds and reports just one floatp box produced.

Essentially the (TYPE FLOAT X Y) declaration is a promise to the compiler that X and Y will hold FLOATP's , so arithmetic may be done unboxed (that is on the value itself, instead of on a pointer to the value, which is the usual case) if possible. The key issue is what is meant by "if possible".

The compiler is conservative. It will perform unboxed arithmetic only on built-in arithmetic functions (PLUS , TIMES, DIFFERENCE, etc), which have unboxed counter parts, and will otherwise box across function boundaries regardless of compiler declarations.

For example:

```
(DEFINEQ (FOOBAR  (N)
        (bind (SETQ X 0.0) (SETQ Y 2.0) for I from 1 to N
                declare (TYPE FLOAT X Y)
                do (SETQ X (FPLUS X (LOG Y)))
                finally (RETURN X))))
```

then

```
(TIMEALL (FOOBAR 100))
```

returns a CPU time of .049 with 601 FLOATP boxes produced (some of which come from the LOG (five per function call)).

Also, the compiler will box in a return context.For example

```
(DEFINEQ (BAR  (N)
        (bind (SETQ X 0.0) for I from 1 to N
                 declare (TYPE FLOAT X )
                 do (SETQ X
                        (PROG ((Y 2.0))
                                (DECLARE (TYPE FLOAT Y))
                                (RETURN (FTIMES Y Y))))
                finally (RETURN X))))
```

then

```
 (TIMEALL (BAR 100 ))
```

returns a CPU time of .022 with 301 FLOATP boxes produced -- notice that BAR seems like it should behave like FOO.

Indeed that is the the greatest drawback of the unboxed arithmetic as it stands now -- it is not always easy to predict what is going to happen -- there are even traps where indiscriminate uses of TYPE FLOAT declarations will actually produce MORE boxes than without them. This is the case if, for example, you use comparison operators (GREATERP, etc) since the compiler boxes each operand before invoking them.

The BAR example may be fixed up as follows:

```
(DEFINEQ (BAR  (N)
        (bind (SETQ X 0.0) for I from 1 to N
                declare (TYPE FLOAT X )
                do (SETQ X
                       (PROG ((Y 2.0) RESULT)
                            (DECLARE (TYPE FLOAT Y RESULT))
                            (RETURN (SETQ RESULT (FTIMES Y Y))))
               finally (RETURN X)))
```

then

```
(TIMEALL (BAR 100))
```

returns a CPU time of .008 with 101 FLOATP boxes produced. Note that the compiler still boxes the result returned by the PROG.

The best way to find out what is happening is to use a combination of TIMEALL and INSPECTCODE . Unanticipated boxing behavior will show up as BOX opcodes -- if you find a sequence of opcodes UNBOX , BOX , function call, UNBOX , then you know you are in trouble. TIMEALL will report the total number of boxes produced.

Basically TYPE FLOAT declarations are best used in tight inner loops of the sort illustrated in FOO.

With all these caveats, I think it is only fair to say that considerable performance inprovements can be realized with judicious use of the TYPE FLOAT declarations; my measurements indicate a factor of ten.

Additional note: TYPE FLOAT vars are by necessity LOCALVARS.

Lyric compatibility note: Allthe entries described for this module are in the Interlisp package. Only the Byte compiler pays attention to TYPE FLOAT declarations -- i.e. use of TYPE FLOAT declarations will be ignored by the XCL compiler.

UNDIGESTIFY

By: Steven C. Bagley (Bagley.pa)

This document last edited on May 27, 1986

INTRODUCTION

This Lafite package allows you to unpack an Arpa Network digest, such as AIList, into its constituent messages. An new item, "Undigest," is placed on the browser menu. When a single message (presumably a digest) is selected, clicking on this item will delete the selected message, and append the constituent messages to the end of the mail folder. If the selected message is not a digest, or is a digest in a format that cannot be parsed properly, then a message will be printed and nothing will happen to the mail folder.

USER OPTIONS

`*DELETE-DIGEST-FLAG*`, if `T` means that the digest message should be deleted if it is successfully parsed. The default is `T`.

`*MOVE-TO-FIRST-DIGEST-MESSAGE-FLAG*`, if `T` means to select the first constituent message, if `NIL` means to select the first undeleleted message after the digest message. The default is `NIL`.

`*DONT-UPDATE-HEADERS-FLAG*`, if `T` means not to copy the To: field from the digest to each constituent message. The default is `NIL`.

PLANNED ENHANCEMENTS

Inserting the contained messages immediately after the digest, rather than appending.

Moving the contained messages to a different mail folder.

NOTES

Many digests are not in the correct format. The parser used in this program tries to be very forgiving, and hence, is relatively slow (about 10 seconds to parse a digest on a Dorado). If everyone adhered to RFC934, the parser could be optimized for speed, but, alas, this is not the case.

UUENCODE

By: Doug Cutting (Cutting.PA@Xerox.COM)

This document last edited on October 7, 1987.

UUENCODE provides facilities for encoding files into printing ASCII characters for transfer by electronic mail. It is compatable with the UNIX™ facility of the same name. For details of the file format see the UNIX™ manual page on ‘uuencode’.

(UUENCODE *FILES INTO-FILE*) [Function]

Encodes the files named by FILES into INTO-FILE. FILES may be either a list or files or a single file name. Note that UNIX™ uudecode does not support multiple files encoded in one file. Thus one should only pass a single file name to UUENCODE if the file is to be decoded under UNIX™. Returns the name of the file written.

(UUDECODE *FILE-OR-STREAM ONLY-ONE-FILE?*) [Function]

Decode from FILE-OR-STREAM writing the decoded files in the connected directory. FILE-OR-STREAM may be either a file name or a stream. If ONLY-ONE-FILE? is non-NIL then only one file will be extracted from FILE-OR-STREAM, and an error will be reported if no encoded file is found. This can be thought of as UNIX™ compatability mode. Returns the list of the names of the files extracted.

(UUENCODE-INTERNAL *INS OUTS DECODE-NAME FILE-MODE*) [Function]

Called by UUENCODE to encode one file. Encodes all bytes from the stream INS to the stream OUTS. DECODE-NAME is the name the file should be given when it is decoded. FILE-MODE is the UNIX™ file mode for the file. DECODE-NAME defaults to (FULLNAME INS) and FILE-MODE defaults to the value of the variable UU.MODE-DEFAULT. Returns OUTS.

UU.MODE-DEFAULT [Global Variable]

The default UNIX™ file mode to encode files under as an integer. UNIX™ uudecode will use this when creating the decoded file. The initial value is 644Q (read & write by owner, read by group and other).

(UUDECODE-INTERNAL *INS ONE-FILE-ONLY?*) [Function]

Called by UUDECODE to decode one file. INS should be a stream open for input. Returns the name of the file extracted or NIL if none is found and ONE-FILE-ONLY? is NIL.

UUENCODE was inspired by Christopher Lane’s BMENCODE package.

VSTATS

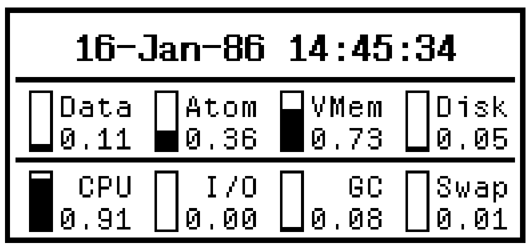
By: Johannes A. G. M. Koomen
(Koomen.wbst@Xerox or Koomen@CS.Rochester)

Uses: SYSTATS, READNUMBER

This document last edited on November 20, 1987

INTRODUCTION

Loading VSTATS will put a VStats entry on the background menu, and execute (VSTATS 'On), which will cause the following display to be created and continuously updated:



DESCRIPTION

VSTATS is a facility for continuously displaying various interesting aspects of a running system. It can display the current time of day, with or without seconds, and/or display memory and disk space utilization, and/or display machine utilization in terms of CPU, I/O, GC and swap time. The display can be regular or inverse-video. The display is updated at user settable intervals, either always or only if the display window is completely visible.

Closing the VSTATS window will remove the background update function.

Left buttoning the VSTATS window causes it to be recomputed and redisplayed entirely. Otherwise display updates only affect those parts that have actually changed, making for a visually quiet and efficient facility.

Middle buttoning the VSTATS window will bring up an Inspector window onto the VSTATS list of options. Left buttoning an option name prints an explanation of the option to the Prompt window. Left buttoning an option value selects it, and middle buttoning an option value presents a menu from which a new value can be selected. The options window looks like this:

VStats Options	
Display.Color	Normal
Update.Always?	No
Show.Disk.Space?	DSK1
MUtil.Hysteresis	20
Space.Panic.Level	Disabled
Clock.Update.Interval	1
Space.Update.Interval	300
MUtil.Update.Interval	1

VSTATS is highly optimized for speed and implemented as a BACKGROUNDFN rather than as a seperate process so as to minimize overhead. As a result, VSTATS can easily be run with display update intervals equal to 1 second.

DETAILS

(VSTATS *on/off*) [Function]

If *on/off* is either ON, On, on, or T, the VStats display is turned on; otherwise off.

VSTATS.CLOCK.INTERVAL [Variable]

If the global variable VSTATS.CLOCK.INTERVAL is a positive number, VSTATS displays an alphanumeric clock (*e.g.*, "1-Aug-85 14:30"), which is updated every VSTATS.CLOCK.INTERVAL seconds. If this interval is less than 1 minute VSTATS displays seconds as well. For those of you who keep their machines running overnight (say, with IDLE or BOUNCE), if the clock display is enabled, VSTATS will resynchronize the local clock with the network daily at midnight. (My machine looses about 15 minutes a week, otherwise!)

VSTATS.SPACE.INTERVAL [Variable]

If the global variable VSTATS.SPACE.INTERVAL is a positive number, VSTATS displays, both graphically and alphanumerically, the utilization of Data, Atom, and VMem spaces and optionally Disk space, which is updated every VSTATS.SPACE.INTERVAL seconds.

VSTATS.SPACE.PANIC.LEVEL [Variable]

If VStats is displaying space utilization, and VSTATS.SPACE.PANIC.LEVEL is a percentage between 1 and 100 (or a fraction between 0 and 1), and any of the memory space utilizations (other than disk) exceed this percentage, VSTATS will flash its window in proportion to the excess, whether the window is occluded or not.

VSTATS.SPACE.SHOW.DISK? [Variable]

If VStats is displaying space utilization, then if VSTATS.SPACE.SHOW.DISK? is non-NIL, Disk space utilization is displayed as well, provided VStats can figure out the total disk size. If VSTATS.SPACE.SHOW.DISK? is T, the default DSK is used, for instance {DSK19} on a Dorado, or {DSK}<LispFiles> on a Dandelion. Alternate Dorado partitions or Dandelion volumes may be assigned to VSTATS.SPACE.SHOW.DISK? as well. If assigned through the options window, VStats will figure out which volumes or partitions are displayable.

VSTATS.MUTIL.INTERVAL [Variable]

If the global variable VSTATS.MUTIL.INTERVAL is a positive number, VSTATS displays, both graphically and alphanumerically, the machine utilization in terms of CPU time, time spent on disk and

Ethernet I/O, garbage collection time, and swapping time, which is updated every VSTATS.MUTIL.INTERVAL seconds.

VSTATS.MUTIL.HYSTERESIS [Variable]

If VStats is displaying machine utilization and VSTATS.MUTIL.HYSTERESIS is a positive number, the relative percentages are based on the average over VSTATS.MUTIL.HYSTERESIS intervals, otherwise they are based on the total time since VSTATS was invoked.

VSTATS.POSITION [Variable]

If the global variable VSTATS.POSITION is a POSITION, the VSTATS display will be put there, otherwise the user is prompted for a POSITION.

VSTATS.BLACK? [Variable]

If the global variable VSTATS.BLACK? is non-NIL, VSTATS displays with inverse video.

VSTATS.ALWAYS? [Variable]

If the global variable VSTATS.ALWAYS? is non-NIL, VSTATS will always update its display when its timers expire, causing its window to come to the top if it isn't already there; otherwise, VSTATS will only update the display if its window is neither partially nor wholly occluded. If it is occluded, VSTATS will, of course, continue to update its internal timers and the display will be updated the first time the timers expire after the display becomes wholly visible again.

Defaults

VSTATS.BLACK? NIL	
VSTATS.ALWAYS?	NIL
VSTATS.POSITION	top right corner of display
VSTATS.CLOCK.INTERVAL	1 second
VSTATS.SPACE.INTERVAL	300 seconds (5 minutes)
VSTATS.SPACE.PANIC.LEVEL	95 %
VSTATS.SPACE.SHOW.DISK?	T
VSTATS.MUTIL.INTERVAL	1 second
VSTATS.MUTIL.HYSTERESIS	20 intervals

If different values are preferred, these variables should be set by the user before loading VSTATS to affect initial display. They can of course be altered anytime using the options menu.

Extras:

A number of functions are required (and supplied) by VSTATS which the author believes might well be part of standard Interlisp-D, *viz*.,

(COVEREDWP *window*) [Function]

Returns T if *window* is partially or completely covered by some other window; NIL otherwise.

(CLOCKTICKS *interval timerunits*) [Function]

Returns the (machine dependent!) number of internal clock ticks over the interval. For instance, on the D'Lion

(CLOCKTICKS 2.5 'MINUTES) = 5211900

(ALTOPARTITIONS) [Function]

On the Dorado, returns a list of partitions set up with an Alto exec, i.e., containing a system boot file. Especially useful with the recently added Extended VMem option, where not all partitions are bootable. Returns NIL on any other machine type. Note: this list of partitions takes between 15-20 seconds to compute.

(DISKUSEDPAGES *dsk recompute*) [Function]

Returns the total number of disk pages in use (complementing DISKFREEPAGES). On Dorado, this is only an estimate, unless *recompute* is non-NIL in which case you wait ~ 8 seconds for the answer.

(DISKTOTALPAGES *dsk recompute*) [Function]

Returns the total number of disk pages available (sum of DISKFREEPAGES and DISKUSEDPAGES).

WALKFILES

Johannes A. G. M. Koomen
(Koomen.wbst@xerox or Koomen@cs.rochester.edu)

October 27, 1989

SUMMARY

WALKFILES is a facility for searching loaded files for arbitrary objects. It complements MasterScope, in that it can do some things that MasterScope can't (like looking for arbitrary substrings in functions); other things Masterscope can do as well or better. WALKFILES does not require an analysis of the files prior to action.

DESCRIPTION

(WALKFILES *pattern file{s} editcommands confirmflg quietflg filepackagetypes*) [Function]

Invokes WALKDEFS on all the objects on each *file{s}* of the types given in *filepackagetypes*. If *file{s}* is NIL, the value fo FILELST is used. If *filepackagetypes* is NIL, all the filepackage types in PRETTYTYPELST are used.

(WALKDEFS *pattern name{s} filepkgtype editcommands confirmflg quietflg*) [Function]

Walks over the *filepkgtype* definition of each name in *name{s}* looking for *pattern*. Pattern can be anything acceptable to EDITFINDP. For each occurrence, first prints the name (unless *quietflg* = *T*) and then prints the occurrence if *editcommands* is NIL, or invokes the editor in interactive mode if *editcommands* is T (asking for confirmation if *confirmflg* = T) , or applies *editcommands* to the name and *filepkgtype* if *editcommands* is a function, or otherwise invokes the editor with *editcommands*.

EXAMPLES

(WALKFILES 'elseif 'MYFILE T) [Example]

Brings up the editor on every definition in MYFILE which contains the symbol elseif.

(WALKFILES 'FOO) [Example]

Prints every occurence of any symbols with FOO as a substring in every definition in any file in FILELST.

(LET (THEFNS) [Example]

```
(LET (THEFNS)
   (WALKFILES '(*ANY* $FOO$ BAR) NIL '(LAMBDA (FN) (PUSH THEFNS FN)) NIL T 'FNS)
   THEFNS)
```

Collects all functions containing either the symbol BAR or any symbols with FOO as a substring on every file in FILELST.

WDWHACKS

By: Atty Mullins (Mullins.pa@Xerox.com)
Revised by: Ron Kaplan (kaplan@parc.xerox.com)

Some short hacks that make window management slightly easier.

Loading this file forces the menu entries: **SLAMWS** (in the background menu) with the subitem **INSPECTORS**, **SEDIT**, and **FILEBROWSERS**, and **POPSHAPE** (under "Close" in the window menu), and replaces the action for **SHAPE** (window menu) with its own call.

When **SLAMWS** is selected, you are asked for a region of the screen, and all of the windows that intersect that region are closed (by call to **CLOSEW**). If one of the **INSPECTORS**, **SEDIT**, or **FILEBROWSERS** subitems is selected, all open (not shrunken) windows of the indicated type are closed.

When **SHAPE** is selected, the old shape of the window is stored, and then selecting **POPSHAPE** will reshape the window to the stored shape. When **POPSHAPE** is called, the current shape is stored, so that doing **POPSHAPE** multiple times will rotate between the two shapes.

The shape/popshape hacks are useful if you want to either get something out of the way temporarily (but not lose it entirely); or to enlarge something temporarily (e.g., for doing a SEE in the typesacript window).

lispusers/WHEELSCROLL

Written by Ron Kaplan, February 2021.

This small file adds the ability to scroll (scrollable) windows by rotating the wheel on a wheel mouse or by moving (2?) fingers on a track pad.

The capability is enabled when WHEELSCROLL.LCOM is loaded.

It is toggled on and off by

(ENABLEWHEELSCROLL ON) (initially (ENABLEWHEELSCROLL T))

The vertical scrolling speed is controlled by the variable

WHEELSCROLLDELTA (initially 20)
The number of points to scroll for each click of the wheel. Higher values give faster scrolling. A negative value reverses the scrolling direction.

HWHEELSCROLLDELTA (initial NIL)
If non-NIL, then this is the delta used for horizontal scrolling.

Implementation:

Lisp receives a key transition on PAD1 or PAD2 for vertical scrolling when the wheel rotates and no other keys are down. (ENABLEWHEELSCROLL T) modifies the keyaction table so that it maps these transitions to characters 156 and 157. Those characters are defined as interrupts that invoke the vertical scrolling action. For horizontal scrolling sideways pushes of a wheel (if it has that) produce transitions on PAD4 and PAD5, which map to interrupt-characters 158 and 159. (156-159 are the highest right-panel characters of character-set 0 that correspond to left-panel control characters, so typically have no other conflicting meaning.)

(ENABLEWHEELSCROLL NIL) causes PAD1, PAD2, PAD4, and PAD5 to be ignored.

Current negative features:

1. When the wheel is depressed for middle-button effect (and no other keys are down), an accidental rotation of the wheel during the transition (up and/or down) may cause unintended scrolling.

We need to develop a strategy, either in Lisp, Maiko, or X, to discriminate intended middle-button pushes from intended scrolling. This is not an issue for track-pad scrolling.

2. When the wheel is rotated over a window that partially occludes a Tedit window with a caret blinking in its unoccluded region, both the target window and the partially obscured Tedit window may scroll.

WHOCALLS

By: Bill van Melle (vanMelle.pa@Xerox.com)

This file contains two useful functions for quick crossreference:

(whocalls *callee usage*) [Function]

maps over all symbols in the current environment, looking for any function that mentions callee according to usage:

values for usage:

USES VAR VARS BOUND USEDFREE GLOBALS
All mean: mention as a variable

NIL CALLS

means calls as a function

(distribute.callinfo) [Function]

inverts all of the call, use global, use free, bound releations for functions, variables from compiled code. Operates by mapping over all symbols in the sysout that are defined as compiled code, and analyzing their definitions. Anything that is called has a CALLEDBY property of all of the things that call it; any variable bound has a BOUNDBY with the list of functions that bind it, variables that are used globally have a USEDGLOBALBY and variables that are used freely have a USEDFREEBY.

(References from interpreted code, etc are not detected, so it isn't 100% guaranteed that if something doesn't have a CALLEDBY that it isn't called.....)

WHO-LINE

By: SML (Lanning.pa@Xerox.com)

INTRODUCTION

Need to know what package you're in? Don't know what your connected directory is? Fret not. The Who-Line is here.

The Who-Line is a window that displays this information on your screen. It is continually updated to reflect the current state of the world (thanks to an entry on BACKGROUNDFNS). Additionally, items in the Who-Line can act as menu items, allowing you to change the state of the machine.

Defining the information displayed in the Who-Line

The values displayed in the Who-Line are determined by the setting of the variable *WHO-LINE-ENTRIES*.

WHO-LINE-ENTRIES [Global Variable]

WHO-LINE-ENTRIES is a list that describes the items that will be displayed in the who-line. Each item in the list should be a list of up to five things: the name of the item; a form that, when evaluated, will produce the value to display; the maximum number of characters in the value; an optional function to call if the item is selected (with the mouse) in the Who-Line; an optional form that will reset any internal state of the entry when evaluated; and an optional string that describes the value displayed by the entry.

[[NOTE: Since the items on the Who-Line are evaluated rather often, it is best if they are fast and efficient (= don't CONS or allocate any space).]]

The following are standard members of *WHO-LINE-ENTRIES*.

WHO-LINE-USER-ENTRY [Variable]

Displays the current user in the Who-Line. Selecting this item in the Who-Line will let you change the logged in user.

WHO-LINE-HOST-NAME-ENTRY [Variable]

Displays the (ETHERHOSTNAME) of the machine you are running on.

WHO-LINE-PACKAGE-ENTRY [Variable]

Displays the package of the current TTY process in the Who-Line. Selecting this item in the Who-Line will let you switch the package of the current TTY process.

WHO-LINE-READTABLE-ENTRY [Variable]

Displays the (name of the) readtable of the current TTY process in the Who-Line. Selecting this item in the Who-Line will let you switch the readtable of the current TTY process.

WHO-LINE-TTY-PROC-ENTRY [Variable]

Displays the name of the current TTY process in the Who-Line. Selecting this item in the Who-Line will let you give the TTY to a different process.

WHO-LINE-DIRECTORY-ENTRY [Variable]

Displays the current connected directory in the Who-Line; the directory is shown in the format "Dir>Subdir>...>Subdir on {Host}". Selecting this item in the Who-Line will let you connect to another directory: the variable *WHO-LINE-DIRECTORIES* (see below) is used to produce a menu of interesting directories. If you are holding down a SHIFT key when you select an item from this menu, the directory name will be COPYINSERTed into the current tty input stream, otherwise you will be connected to that directory.

WHO-LINE-VMEM-ENTRY [Variable]

Displays the percentage of the VMem file that is currently being used in the Who-Line. If the VMem file is inconsistant, the number will be preceeded by an asterik ("*"). Selecting this item in the Who-Line will let you do a (SAVEVM).

WHO-LINE-SYMBOL-SPACE-ENTRY [Variable]

Displays the percentage of symbol space that is currently in use.

WHO-LINE-TIME-ENTRY [Variable]

Displays the current time in the Who-Line. Selecting this item in the Who-Line will let you do a (SETTIME). If you hold down a shift key when you select this item, the current time will be COPYINSERTed into the current tty input stream instead.

The default value of *WHO-LINE-ENTRIES* contains all these items

Other ways to tailor the Who-Line

WHO-LINE-ANCHOR [Variable]

WHO-LINE-ANCHOR describes where the who-line will be displayed. If *WHO-LINE-ANCHOR* contains the symbol :TOP, the Who-Line will be anchored at the top of the screen; if it contains the symbol :BOTTOM it will be anchored at the bottom of the screen. If *WHO-LINE-ANCHOR* contains the symbol :LEFT, it will be anchored to the left side of the display; if it contains the symbol :CENTER it will be centered on the screen; if it contains the symbol :JUSTIFY it will run the width of the screen; if it contains the symbol :RIGHT it will be anchored to the right side of the screen. Finally, if *WHO-LINE-ANCHOR* is a POSITION, it will be used as the lower left corner of the Who-Line. The default value is (:CENTER :BOTTOM).

WHO-LINE-NAME-FONT [Variable]

The font used to display the names of the items in the who-line. The default is HELVETICA 8 BOLD.

WHO-LINE-VALUE-FONT [Variable]

The font used to display the values in the who-line. The default is GACHA 8.

WHO-LINE-COLOR [Variable]

The color of the Who-Line. Legal values are the keywords :WHITE and :BLACK. The default is :WHITE.

WHO-LINE-BORDER [Variable]

The border width of the Who-Line window. The default is 2.

WHO-LINE-TITLE [Variable]

The title of the Who-Line window. The default is NIL.

WHO-LINE-DISPLAY-NAMES? [Variable]

If *WHO-LINE-DISPLAY-NAMES?* is true, the names of items in the who-line will be displayed; otherwise they will not be shown. The default value is T.

WHO-LINE-UPDATE-INTERVAL [Variable]

The number of milliseconds between updates of the who-line. The default is 100 milliseconds.

Installing new Who-Line options

Changing the above variables has no direct effect on the who-line. These values need to be installed in the Who-Line before they can take effect.

(INSTALL-WHO-LINE-OPTIONS) [Function]

INSTALL-WHO-LINE-OPTIONS installs the above options in the Who-Line, and updates the Who-Line accordingly.

The Who-Line supports an easy way to interactivly add or remove entries. If you click on the Who-Line while holding down the EDIT or CONTROL key, you will be given a chance to add or remove items from the Who-Line.

WHO-LINE-ENTRY-REGISTRY [Global Variable]

A list of all known Who-Line entries. This is used to construct the menu of possible new entries for the Who-Line.

Who-Line process state

The who-line entry *WHO-LINE-TTY-STATE-ENTRY* tries to display the current state of the TTY process.

WHO-LINE-TTY-STATE-ENTRY [Variable]

A Who-Line entry that displays the "state" of the current TTY process in the Who-Line. The typical state of a process is the name of the function that is currently running in that process. This simple minded result can be altered by use of the following items.

[[NOTE: Because of the nature of the Lisp scheduler, this information is almost always out of date.]]

The Who-Line "state" can be explicitly controlled from code. If the special variable *WHO-LINE-STATE* is bound, its value is taken to be the state of that process. You can use this feature to provide visual indiation of the state of your code by using the programming idiom:

```
(LET ((*WHO-LINE-STATE* indicator))
  (BLOCK)                       ;Give the Who-line a chance to run
  ...your-code...)
```

This will run the *...your-code...* with the Who-Line state of the process set to (the value of) *indicator*. The call to BLOCK insures that the Who-Line has a chance to update before *...your-code...* is run.

WHO-LINE-STATE-UNINTERESTING-FNS [Global Variable]

If there is no declared who-line state (via a WITH-WHO-LINE-STATE form), then the name of the function that is currently running is used as the who-line state. However, if the function is on the list *WHO-LINE-STATE-UNINTERESTING-FNS*, the function that called *it* is used instead. The default value of *WHO-LINE-STATE-UNINTERESTING-FNS* is (BLOCK AWAIT.EVENT).

WHO-LINE-STATE [Property]

If the function that is currently running has a WHO-LINE-STATE property, the value of that property is used as the who-line state. This is used to convert functions like \TTYBACKGROUND to meaningful values like "TTY wait".

(WHO-LINE-REDISPLAY-INTERRUPT) [Function]

Updates the Who-Line. It is intended that this function be installed on an interrupt character, so that the user can easily force an update of the Who-Line. For example,

```
(ADVISE 'CONTROL-T 'BEFORE '(WHO-LINE-REDISPLAY-INTERRUPT))
```

will cause a ^T interrupt to update the Who-Line as well as its current behavior of printing state information in the Prompt window. Alternatly, you can define a new interrupt character that will force an update of the Who-Line;

```
(INTERRUPTCHAR (CHARCODE ^U) '(WHO-LINE-REDISPLAY-INTERRUPT) 'MOUSE)
```

will cause the Who-Line to be updated whenever the user hits a ^U.

Other interesting things

WHO-LINE-DIRECTORIES [Global Variable]

A list of interesting directories used to generate a pop-up menu of directories to connect to when you select the DIRECTORY item in the Who-Line. The default value is a list containing just your LOGINHOST/DIR. When the Who-Line notices that you have changed your connected directory, it updates this list to contain the new directory.

(CURRENT-TTY-PACKAGE) [Function]

Returns the name of the package of the current TTY process. This function is used in the default value of *WHO-LINE-ENTRIES*.

(CURRENT-TTY-READTABLE-NAME) [Function]

Returns the name of the readtable of the current TTY process, or the string "Unknown" if it can't figure out the name. This function is used in the default value of *WHO-LINE-ENTRIES*.

(SET-PACKAGE-INTERACTIVELY) [Function]

Pops up a menu of currently defined packages. If the user selects one of them, the current package is changed to the selected package.

(SET-READTABLE-INTERACTIVELY) [Function]

Pops up a menu of currently known readtables. If the user selects one of them, the current readtable is changed to the selected readtable.

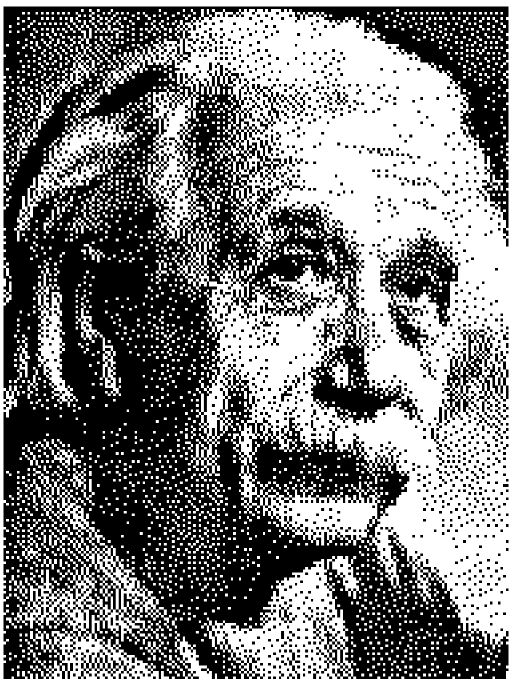
WINK

By: Larry Masinter (Masinter.pa@Xerox.com)

This is a file containing bitmap demos. To bring up a bitmap of Marilyn Monroe winking, type:

(MARILYN)

This file also has bitmaps EINSTEIN and LINCOLN:



"AL"

"ABE"

WORDFNS

By: Ron Kaplan (Kaplan.pa@Xerox.com)
Becky Burwell (Burwell.pa@Xerox.com)

Uses: SETSTRINGLENGTH

This document last edited on August 19, 1988.

INTRODUCTION

WORDFNS is a set of functions for manipulating files of words. There are functions to do the following: sort files, manipulate sorted files, provide common i/o functions for word files, provide mapping and translation mechanisms, provide common translation functions, and provide packaged mapping utilities.

The idea behind the mapping mechanism is that you can translate a file or list of files by specifying a read function to operate on each chunk of a file (the obvious two chunks are words and lines). You can specify file specific translation functions, default functions (when file specific functions are not provided) and common translation functions for all files. The input to the first translation function is the result of applying the read function to an input stream open on a file. The output of the first translation function is passed as input to the second translation function, etc.

USE

Note: for any file, if NIL or T is specified then the results are printed in the executive window.

Sorting Files

(SORTWORDFILE *IFILES OFILE COMMONTRANSFNS DEFAULTTRANSFNS READFN COMMONCOMPAREFN KEEPDUPLICATES FIELDS SEPARATOR REVERSEORDERFLG FASTFLG*) [Function]

The functions sorts the words on IFILES and stores the result back on OFILE. Th duplicates are eliminated unless *KEEPDUPLICATES* is non-NIL. For a description of the function of the argumentsCOMMONTRANSFNS, DEFAULTTRANSFNS and READFN see the section entitled "Translation Mechanisms". The argument *FIELDS* is used to specify the sorting order. The separator of the fields is specified in *SEPARATOR*. If *REVERSEORDERFLG* is T the result of the sort is reversed. *FASTFLG* set T causes the sort to caches the fields by consing allowing for a quicker sort (but consumes memory).

FIELDS is one of: NIL, a list of field numbers or else a list of one, two or three element lists of the form: FieldNumber Type CompareFn where type is either STRING (the default) or NUMBER. The default comparefn for STRING is ALPHORDER; for NUMBER is NUMORDER [Argument]

SEPARATOR is one of the following: a character string, a bittable, a list of single character atoms or numbers or one of the special atoms WHITESPACE (indicating a space or tab) or the atom TAB. The default is WHITESPACE.

[Argument]

>> should I put NUMORDER and GetNthField here?<<

Note: two related functions, NUMORDER and GetNthField, are described in the miscellaneous section.

Functions for use with sorted files

In each of the following functions:

COMMENTFILE contains the details of the result of the function (for example, the number of strings that were read in from each file) [Argument]

(COMMONSORTEDFILES *file1 file2 ofile COMMENTFILE*) [Function]

Computes the intersection of two sorted files, *file1* and *file2* and the results are stored on *ofile*. The files are read a line at a time. The value is the full name of *ofile*.

(COMPARESORTEDFILES *file1 file2 ofile IMINUS2 2MINUS1*
COMMENTFILE COMMENT) [Function]

The two sorted files, *file1* and *file2*, are compared a line at a time. The common lines are stored on *ofile*. The output is in two colums: the left column for those lines in *file1* that do not exist in *file2* and the right column for those lines in *file2* that do not exist in *file1*. The two flags *IMINUS2* and *2MINUS1* are used to determine how the the comparisons will be performed. If they are not specified they are both assumed to be T thus meaning that the comparison will be performed by subtracting *file2* from file *file1* and *file2* with *file1* substracted. If only one of *IMINUS2* or *2MINUS1* is specified then only the specified one way comparison will be done. *COMMENT* is intended to be a string which, by default, is the string "Comparison". This string is inserted at the top of the file. The value is the full name of *ofile*.

(DIFFSORTEDFILES *FILE1 FILE2 OUTFILE COMMENTFILE*) [Function]

The result of subtracting *FILE1* from *FILE2* is stored on *OUTFILE*. The files are read a line at a time. The value is the full name of *ofile*.

I/O Functions

In the two major read functions, DREADLINE and DREADWORD, the SPACE, CR, LF and ^Z in any character set are interpreted to be the corresponding character in character set zero.

(DREADLINE *stream string skipsemicolons*) [Function]

Words are read from the word-stream *stream*, smashing them into *string*, which grows as needed. Returns NIL at EOF. Skips leading and trailing separators, and if *skipsemicolons* is non-NIL then sequences from ";" to EOL are treated as a composite separator or end-marker. Unlike DREADWORD (desribed later in this section), segments are separated only by EOL, so compounds are not split into components. Note that *stream* must have been set up so that BIN/READCCODE returns NULL on EOF.

(DREADLINESKIPSC *stream string skipsemicolons*) [Function]

Calls DREADLINE with *skipsemicolons* bound to T.

(DREADWORD *stream string*) [Function]

Words are read from the word-stream *stream*, smashing them into *string*, which grows as needed. Returns NIL at EOF. Skips leading and trailing separators, and treats sequences from ";" to EOL as a composite separator or end-marker. Unlike DREADLINE, segments are separated by space as well as EOL, so splits compounds into components. Note that *stream* must have been set up so that BIN/READCCODE returns NULL on EOF.

(INPUTWORDSTREAM *FILE NOPRINT*) [Function]

Returns a stream that is guaranteed to be open for word-reading (e.g. using DREADLINE or DREADWORD) at the beginning of *FILE*. If *NOPRINT* is NIL then the fullname of the file will be output.

(OUTPUTWORDSTREAM *FILE*) [Function]

Returns and opens a stream for the output of words (sequential text) guaranteed closed when reset context is exited and deleted if there is an error.

Translation Mechanisms

Translation mechanisms are supplied to allow great flexibility in translating one or more files which may be in different formats and have unique translations applied to them. To specify how a file is to be read a read function (READFN) can be specified. Two common read functions, described previously, are DREADLINE and DREADWORD.

As mentioned earlier each file can have unique or common translation functions. A translation function is a function which takes two arguments: a string (the input to be translated) and an optional scratch string which can be destructively modified. The output of the translation function is one of the following: a string, the value T or the value NIL. Readers may wish to refer to the LispUsers module SETSTRINGLENGTH. The special value T denotes that the output is the same as the input. A value of NIL means that nothing will be kept.

>> a better name for translation set <<

>> what is the syntax?<<

IFILES is either a single file name, a single translation set, a list of file names or a list of translation sets. Translation sets have the form: (READFN *READFN TRANFN1 TRANSFN2* ...). (Note the first element of the list is the actual atom READFN and the second element [Argument]

READFN is the read function that is used for reading the files. It is passed a stream. Unless specified otherwise, the default read function is DREADWORD. [Argument]

DEFAULTTRANSFNS is a single function or list of functions which is first applied to the first file. What is given to the translation function is determined by what the read function passed. Unless specified otherwise, the default read function is DREADWORD. The result of applying the first function in DEFAULTTRANSFNS is input to the second function in DEFAULTTRANSFNS. This result is then passed on for application to the functions in COMMONTRANSFNS. The special value T denotes that the output is the same as the input. [Argument]

COMMONTRANSFNS [Argument]

DONTPRINT is the argument which decides whether the details of the translation functions will be printed. By default it is NIL meaning that the details will be printed. [Argument]

(TRANSLATEWORDFILE *IFILES COMMONTRANSFNS DEFAULTTRANSFNS READFN DONTPRINT*) [Function]

TRANSLATEWORDFILE produces an output file by translating each word in (possibly a list of) *IFILES* through a translation function. List elements of files are paired with their own idiosyncratic translation function. Otherwise the *DEFAULTTRANSFNS* is used. *COMMONTRANSFNS* are applied to the results of the default or file-specific translations to produce the translation string. If any translation function returns NIL, that string is skipped. A translation function is assumed to be an identity if it returns T, which makes simple predicates easy.

(COLLECTWORDFILE *IFILES COMMONTRANSFNS DEFAULTTRANSFNS READFN DONTPRINT*) [Function]

Returns the list of non-NIL values of functions applied to words in *IFILES*.

(MAPWORDFILE *IFILES COMMONTRANSFNS DEFAULTTRANSFNS MAPFN READFN DONTPRINT*) [Function]

Maps mapping function over words in *IFILES*. Nothing is setup for output.

Packaged Mapping Utilities

(LONGESTWORDS *FILES COMMONTRANSFNS DEFAULTTRANSFNS READFN DONTPRINT*) [Function]

The list of longest translated words in *FILES* is returned.

(SEXPRCOUNT *FILE RDTBL*) [Function]

Returns the number of s-expressions in *FILE* using *RDTBL* to read.

(WORDCOUNT *IFILES COMMONTRANSFNS DEFAULTTRANSFNS READFN DONTPRINT*) [Function]

The total number of translated words in *IFILES* is returned.

(FINDPREFIXES *IFILES OFILE PREFIXES BUTNOT READFN*) [Function]

FINDPREFIXES produces an output file *OFILE* of those strings read by *READFN* from *IFILES* which match at least one prefix in the list of prefix strings *PREFIXES* and do not match any prefixes in the list of prefixstrings *BUTNOT*.

(FINDSUFFIXES *IFILES OFILE SUFFIXES BUTNOT NOCAPS READFN*) [Function]

FINDPREFIXES produces an output file *OFILE* of those strings read by *READFN* from *IFILES* which match at least one suffix in the list of suffix strings *SUFFIXES* and do not match any suffixes in the list of suffix strings *BUTNOT*. If *NOCAPS* is specified then the match succeeds if the string does not have the first letter capitalized.

(FINDSUBSTRINGS *IFILES OFILE SUBSTRINGS READFN*) [Function]

FINDSUBSTRINGS produces an output file *OFILE* of those strings read by *READFN* from *IFILES* which match at least one substring in the list of substrings *SUBSTRINGS*.

Translation Functions

(MIXEDCASEP *W*) [Function]

Returns W if it contains mixtures of uppercase and lowercase characters after the initial character.

(PROPERP *W*) [Function]

Returns T if the first characters of W is uppercase.

(NOTPROPERP *W*) [Function]

Returns T if the first characters of W is not uppercase.

(REVERSESTRING *W STR*) [Function]

Reverses W into STR and returns STR.

Examples

This example will printout all the lines that have either the prefix "re" or "no" but not the prefix "non".

```
(FINDPREFIXES '{dsk}Myfile T '("re" "no") '("non") (FUNCTION DREADLINE))
```

This example will output to file {Phylum}<Project>Suffixes all the words in the files {dsk}File1 and {dsk}File2 that end in "ion" that do not have the first letter capitalized.

```
(FINDSUFFIXES '({dsk}File1 {dsk}File2) '{Phylum}<Project>Suffixes "ion" NIL
T (FUNCTION DREADWORD))
```

Miscellaneous Functions

(GETNTHFIELD *STRING N SEPARATOR FIELDTYPE*) [Function]

The Nth field in STRING is returned using *SEPARATOR* as the field separator and the field type is coerced to type *FIELDTYPE*.

N is a simple positive integer [Argument]

SEPARATOR is the same as that for SORTWORDFILE [Argument]

FIELDTYPE is either the atom NUMBER or the atom STRING and indicates how the type that the field should be coerced to. The default *FIELDTYPE* is STRING.

Example

(GETNTHFIELD "So long and thanks for all the fish" 4 ’WHITESPACE ’STRING)

returns the string "thanks".

(GETNTHFIELD "Joe Smith/5551212/12 Pleasant Lane/" 2 "/" ’NUMBER)

returns the integer 5551212.

(DCOPYSTRING *W STR*) [Function]

Copies string *W* into string *STR* and returns *STR*.

(NUMORDER *NUMBER1 NUMBER2*) [Function]

Returns >>??<<

Example

```
(SETQ  MyString (ALLOCSTRING 1))

(DCOPYSTRING "This is a much longer string" MyString)
```

XCL-BRIDGE

By: Jan Pedersen (pedersen.PA @ Xerox.com)

XCL-BRIDGE is a module that assists in the transformation of ascii Common Lisp source files to Lisp managed files and vice versa. In the text-to-managed-file direction, user interaction is employed to repair read-in forms before establishing a resident image of a Lisp managed file. In the managed-to-text-file direction, a few simple transforms are employed to translate common filepackagecoms to equivalent Common Lisp forms.

All entry points are external to the "XCL" package.

(XCL:TEXT-TO-MANAGED-FILE *pathname filename* &key *(package* "USER"*)*
(readtable "XCL"*) (read-base* 10*) (combine-comments* t*)*) [Function]

Reads an ascii lisp source file named by "pathname" and converts it to a managed file with rootname "filename". The package, readtable, and read-base employed to read the ascii file may be specified via keywords arguments, or defaulted, as shown. If the reader environment arguments are defaulted and the source file has a emacs-style "mode line", then the package and read-base will be as indicated by the "mode line". The "combine-comments" keyword controls whether adjacent comments at the same ";" level should be combined when generating sedit-style comments for the converted file.

Note that forms are only read from the ascii lisp source file, not evaluated. It is assumed that the converted file should be made (via "il:makefile") and compiled before any evaluation should be attempted.

Text-to-managed-file proceeds incrementally and interactively to convert the specified file. First all the forms are read, and presented to the user for editing (via Sedit). If the user accepts this primary phase, a filecoms is generated and again, presented to the user for editing. If the user accepts the generated filecoms, a file (and its contained definitions) is instantiated, completing the conversion.

(XCL:MANAGED-TO-TEXT-FILE *filename pathname* &key *(package* "USER"*)*
(readtable "XCL"*) (print-base* 10*)*) [Function]

Prints a managed file, with rootname "filename", whose source definitions must be resident, to an ascii file "pathname" in a form suitable for reading by any Common Lisp reader. The read-print environment of the managed file may be overwritten via the keyword arguments "package", "readtable", and "print-base". Many Interlisp "filepackagecoms" are translated to their Common Lisp equivalents. For example, "il:declare\:" forms are transformed to "eval-when" forms and "il:files" forms are transformed to "require" forms. As an additional convenience, defdefiners are printed as equivalent defmacros.

Subject: Converter for Xerox Bitmaps to X bitmaps.
To: Lispusers^.x
cc: Rao.pa

{ERINYES}<LispUsers>medley>xerox-to-xbm.lisp
{ERINYES}<LispUsers>medley>xerox-to-xbm.dfasl

I assume soon to trick to PHYLUM.

One function

(xerox-to-xbm outpath xerox-bm &optional name)

Generates a bitmap in the "include" format used by X applications. You can test these by making them into your background in X with:

xsetroot -bitmap outpath

-- Ramana

XORcursorPatch

By: Christopher Lane (Lane@sumex-aim.stanford.edu)

This document last edited on July 22, 1987

INTRODUCTION

This module allows the 1186/Daybreak (only) users to twiddle the hardware bits so that they can have an inverting cursor (white on black and black on white instead of black on everything) and provides a patch to keep the system from undoing the effect when calling VIDEOCOLOR to reset the screen.

USE

(DOVE.XOR.CURSOR *FLG*) [Function]

The argument *FLG*, if T, will switch to the inverting cursor mode. If *FLG* is NIL it will switch back to normal mode. If *FLG* is a number between 0 and 15 then it is used as the 'mix-in rule' and has an effect according to the table below.

Mix-in rules

	Screen	Source	Cursor Mode
0	All Black	None	
1	Normal	Normal	Paint
2	Normal	Inverted	Paint
3	Normal	None	mode.
4	Inverted	Normal	Erase
5	All Black	Inverted	Paint
6	Inverted	Normal	Invert
7	Normal	Inverted	Erase
8	Inverted	Inverted	Erase
9	Normal	Normal	Invert
10	All White	Inverted	Paint
11	Normal	Normal	Erase
12	Inverted	None	
13	Inverted	Inverted	Paint
14	Inverted	Normal	Paint
15	All White	None	

This table is relative to the normal mode of the display (1), normal screen, normal cursor in paint mode. The inverted display, (VIDEOCOLOR T), would be 13, inverted screen and inverted cursor in paint mode. There is probably a more precise or logical way to notate these modes, but this should give you a rough idea ofwhat's available

Note

The function is set up such that when trying different modes, you must do a (DOVE.XOR.CURSOR) (no argument) between calls.

No warranty expressed or implied, but we have been using it locally without problem (at least as far as I know). Enjoy.

>>#<<. CLOS BROWSER

One of the key components of CLOS in inheritance. The CLOS Browser provides functionality for displaying this structure and for extending it. It also provides functions for displaying and changing the class definitions and method definitions which make up a system written in CLOS.

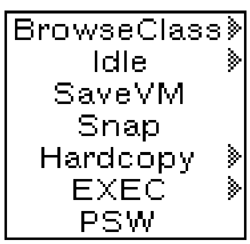
Creating a Browser

A browser can be createded in two ways:

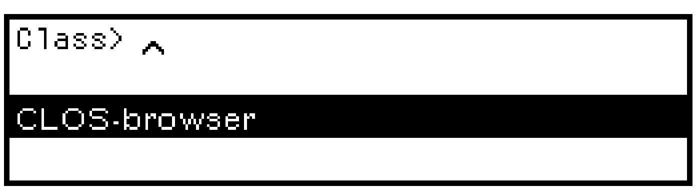
- Via a menu option from the Background Menu
- By calling the function CLOS-BROWSER:BROWSE-CLASS on a class

Creating a browser via the Background Menu

When the CLOS-BROWSER module is loaded, an enty is added to the Background Menu, as shown below:



Selecting the menu item BrowseClass brings up a window, with a prompt for the name of the class to use as the root of the browser as shown below.



Type in the name of the class you wish to browse at the flashing cursor, and the class graph will be drawn in the window.

Creating a browser programmatically

Browsers can also be created by calling the function BROWSE-CLASS:

(BROWSE-CLASS *&OPTIONAL CLASS-NAME-OR-LIST &KEY :WINDOW-OR-TITLE :GOOD-CLASSES :POSITION***)** **[Function]**

This function brings up a browser on the class named or the list of classes named. If a window is supplied for the *:WINDOW-OR-TITLE* argument, then the browser is created in that window, else an appropriately sized window is created. The window is positioned at the *:POSITION* argument or, if not supplied, then the position is set via the mouse. If a text string is supplied for the *:WINDOW-OR-TITLE* argument, then that string is used for the window title, else the string "CLOS-browse" is used. If *:GOOD-CLASSES* is supplied, then only those classes in the list are displayed.

Using the Class browser

Instances of CLOS-BROWSER are operated on through a mouse-based interface.

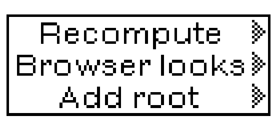
Buttoning on the browser will cause one of the following menus to be popped up:

- One menu appears when the left or middle button is pressed while the mouse is in the title bar. This menu has operations that apply to the browser itself.
- The other menu appears when the middle button is pressed when the mouse is on one of the nodes in the browser.

If the left button is pressed when the mouse is on a node, that node is boxed. This marks the node for some operations.

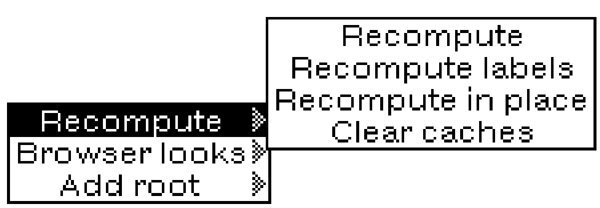
Options in the title bar menu

The following menu appears when you left- or middle-button in the title bar.



Recompute and it’s suboptions

Selecting the Recompute option and dragging the mouse to the right causes the following submenu to appear:

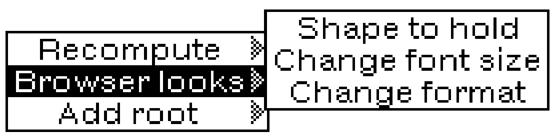


Most of these items change the appearance of the browser, not the contents.

Recompute	Recomputes the browser from the starting objects. It does not recompute the labels for each node if those labels are cached in the Label-Cache slot of the browser.
Recompute Labels	Recompute the browser from the starting objects, including the labels.
Recompute inPlace	Recompute the browser without affecting the scrolled location of the lattice within the window.
Clear caches	Clear the caches of the nodes.

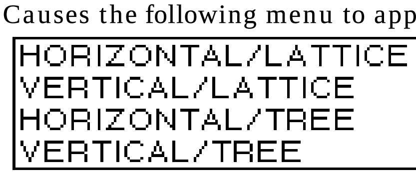
Browser looks and it’s suboptions

Selecting the Browser looks menu item and sliding to the right causes the following submenu to appear:



Selecting one of these options changes the looks of the browser.

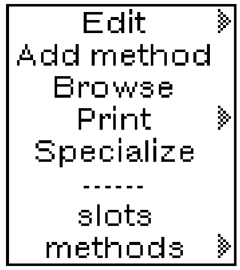
Shape to hold	Make the window for the browser just large enough to contain the browser.
Change font size	Causes a menu of alternative font sizes to pop up. Selecting one of these causes the browser to be redrawn with the nodes at that font size.
Change format	Causes the following menu to appear:



Horizontal/Lattice	Lays out the grapher as an horizontal lattice.
Vertical/Lattice	Lays out the grapher as a vertical lattice.
Horizontal/Tree	Lays out the grapher as a horizontal tree.
Vertical/Tree	lays out the grapher as a vertical tree.

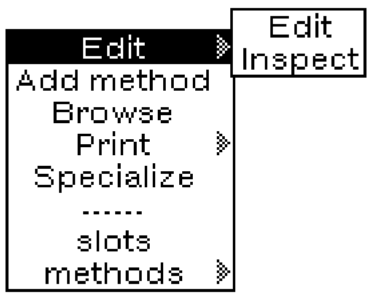
Options in the Middle-button menu

The following menu appears when you middle-button over a node in the graph:



Edit and it's suboptions

Selecting Edit causes an editor on the class definition to be brought up. Sliding the mouse to the right causes the following menu to appear:



Edit	Edits the class named by the node
Inspect	Inspects the class object named by the node.

Add Method

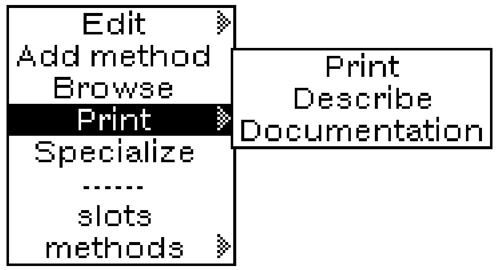
Selecting the Add Method option brings up an editor window with a template for a method to be added to that class. When the editor is done the method is installed for that class and the menu updated.

Browse

Selecting the Browse option causes a browser to be created starting with that class as the root.

Print and it's suboptions

Selecting Print prints out the class definition. Sliding the mouse to the right causes the following menu to appear:



Print — Print’s the class definition

Describe — Describes the class, listing it’s metaclass, it’s supers classes, it’s subclasses, it’s CPL, and the number of methods specialized to it.

Documentation — Print’s the documentation string for the class

Specialize

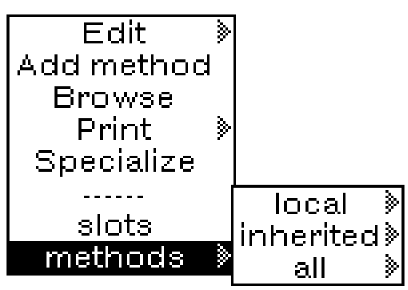
Selecting the Specialize option brings up an editor window with a template for a subclass to be added to that class. When the editor is done the class is installed and the browser updated.

Slots

Selecting the Slots option is the same as selecting the Edit option, it brings up an editor on the class definition.

Methods

The Methods option allows you to edit one of the methods defined for that class. Selecting it and sliding to the right brings up the following sub-menu:



Local — Bring up a menu of the local methods, ie methods directly defined for this class

Inherited — Bring up a menu of the methods this class inherits from it’s superclasses.

All — Bring up a menu of all the methods defined for this class, both local and inherited.

Selecting an item with the left button from the resulting menu brings up an editor on that method. If there are multiple methods that apply, a gray triangle appears in the right edge of the menu next to that item. Sliding to the right brings up a menu of method specializers to select the appropriate method.

This directory contains fonts and font information which are to be used for Xerox internal uses only. Under no circumstances can these fonts be released for customer use. For information, contact Frank Shih, Lisp Development, Xerox Artificial Intelligence Systems.

The screen fonts labelled ITCBauhaus are in fact just renamed copies of the font Modern. This is because ITCBauhaus is not yet available at 72 dpi, and so the generic Modern is substituted instead. Printers containing the font ITCBauhaus should be able to correctly render the file, however.

HostUp

By: Johannes A. G. M. Koomen
(Koomen.wbst@Xerox or Koomen@CS.Rochester.edu)

This document last edited on December 27, 1988

SUMMARY

This module provides the function HOSTUP? which attempts to find out if a given host is currently available. Contrast this to the system function HOSTNAMEP, which returns T if it has *at any time* successfully opened a connection to given host, whether or not it is currently available.

DETAILS

(HOSTUP? hostname) [Function]

Returns T if and only if the given host is currently responding. No distinction is made between dead and non-existing hosts.

HOSTUP.TIMEOUT [Global variable]

The function HOSTUP? returns NIL if no response is received from the given host within HOSTUP.TIMEOUT milliseconds. Default value is 15,000.

HOSTUP.RETRYCNT [Global variable]

This variable indicates the number of times the function HOSTUP? sends requests to the given host. Each time through the loop the function waits longer by a geometrically increasing amount if time, such that the total time does not exceed HOSTUP.TIMEOUT. Default value is 5. Hence, in the default case, a call to HOSTUP? with a dead host ends up sending a request to the host 5 times, waiting for an answer about 500, 1000. 2000. 4000 and 8000 milliseconds, respectively.

IRIS

By: Nuyens (Nuyens.pa@Xerox.Com)

Requires: COLOR and COLOROBJ

This document last edited on January 31, 1987.

INTRODUCTION

The LispUsers module IRIS is a collection of functions for using an IRIS (Integrated Raster Imaging System) with Interlisp-D. The IRIS is a sophisticated 3-d color graphics workstation produced by Silicon Graphics Incorporated. Much of its imaging system is implemented in special purpose VLSI, making it a very powerful graphics engine.

IRIS provides three separate ways to exploit the IRIS in Interlisp-D. The first is to use the IRIS as an Interlisp-D imagestream. In this capacity the IRIS serves as the output device for a program which needn't even know that it is outputting to the IRIS. Thus standard sytem utilities (such as GRAPHER, SKETCH, and TEDIT) can make use of the IRIS without change. The images produced by these system utilities can then be manipulated with the special abilities of the IRIS (3-d rotation, color updates, double buffering, object definitions, etc.)

The second mode of use for the IRIS is to directly access the IRIS graphics library. In this mode, Interlisp-D provides stubs for the functions in the IRIS graphics library. Thus, Interlisp-D provides the illusion of having the IRIS as a direct device like the standard screen. However, it is actually a separate computer accessible across the 10-MB Ethernet.

The third mode is to use the view controller to interactively view scenes created with either of the two modes. The view controller allows the user to create an object once (interactively or via program) but then view the object many times on the IRIS without recreating the object. This object downloading allows real-time rotation of the object displayed on the IRIS.

This package also implements boot service for the IRIS workstation. The D-machine supplies the boot program to the IRIS over the Ethernet, eliminating the need for a floppy-disk on the IRIS.

This package benefited substantially from comments and improvements by Stu Card, Michel Desmerais and Lennart Lovstrand.

FILE DESCRIPTION

The implementation includes the following files:

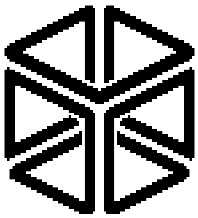
LOADIRIS	after loading this file, the user will be prompted to place the IRIS icon. Buttoning this icon will produce a menu including the choice "create loadup panel". Choosing this item will produce a panel which can load the IRIS files. Button the "Standard" entry in the loadup panel, then button on the small SGI logo next to "Load". Prior to being able to run any applications, the IRIS must be running the terminal program. This can be booted from the floppy on the IRIS or from the Lisp workstation. To use the workstation as a bootserver, the boot file for the IRIS terminal program (see msg below) must be copied to {DSK}irisbootfile (or a directory in the list IRISBOOTDIRECTORIES) on the Dandelion that is to be the bootserver. At that point, hit the reset button on the IRIS and type 'n' to the monitor prompt on the iris. That will initiate a net boot from the Dandelion.
IRISSTREAM	The file which contains the Interlisp-D imagestream definition for the IRIS. Thus, the standard Interlisp-D graphics facilities can be applied to an open IRIS imagestream. After this is loaded, images which are processed through the standard hardcopy menu can be sent to the IRIS. "Iris" will appear as a printer when the "to a printer" choice is made from the roll-off menu entry "Hardcopy".
IRISLIB	this file contains the stubs for the functions in the IRIS library. For instance, corresponding to the function circf (found in the IRIS documentation), there is a function IRIS.CIRCF . The arguments are as listed in the IRIS documentation with the addition of the argument SPPstream. This is the SPP connection opened to the IRIS. (If omitted, it defaults to the value of IRISCONN, which is set by OPEN.IRISCONN). Where an argument is a matrix, it is passed to the lisp function as a list of the rows, each row itself a list. When a library fn returns a matrix, the appropriate (usually floatp) matrix is passed in and the elements are set.
IOLIB	this file contains the communication primitives which are used by the fns on IRISLIB.
IRISNET	this file contains the network support for the iris. The IRIS must be on the same network as the D machine (the IRIS doesn't handle routing through the gateway properly). If the IRIS is being bootserved from the Lisp workstation, the boot server software will set IRISNSHOSTNUMBER automatically. (The diagnostic messages can be inhibited by setting \IRIS.VERBOSE to NIL. They are left in because they can be useful when initially getting the IRIS to boot).
IRISVIEW	This file contains the functions supporting the interactive viewing menu, together with the object definition facility it uses.
IRISDEMOFNS	This file contains an example 3-d function for the IRIS. It is a function called TETRA which draws a colored, 3-d recursive tetrahedron.
IRIS.TEDIT	This file.

GETTING STARTED

N.B. All variables and functions in this document are written as in the old Interlisp readtable. To type in the examples in this document, bring up an "Old-Interlisp" Exec. (available from the background menu by rolling off "exec" and then "Interlisp".)

-1) type (FILESLOAD LOADIRIS)

It will ask you to position the following (SGI) icon:

 (it can be recreated later by typing (IRIS.CREATE.ICON) at the Interlisp executive).

0) Next type (FILESLOAD IRISSTREAM IRISVIEW)

1) Boot the IRIS

If you have a floppy drive for your IRIS, use it to boot the IRIS GL2 (XNS) Terminal program. (If your IRIS is a workstation, not a terminal, you need to run the WSIRIS program available from SGI).

If you do not etherboot, you must set the variable IRISNSHOSTNUMBER. (see variables section below.)

If you have no floppy, you are going to "etherboot" the IRIS. To begin ensure that IRISBOOTDIRECTORIES is set properly. (see Magic Variables section below).

Press a mouse button in the SGI icon. Choose "Enable bootserver". Now boot the IRIS(by pressing the black button labelled "reset" on the IRIS display). To the prompt on the IRIS, type (on the iris keyboard) N (followed by carriage return). To the eventual prompt "connect to which host?" type < (carriage return). Etherbooting will automatically set the variable IRISNSHOSTNUMBER. Status messages will be printed to the promptwindow. The IRIS will beep when it is finished booting.

2) Open an IRIS stream

Button in the IRIS stream and choose "Create IRISView panel". Confirm the command by clicking left button of the mouse. "Connected" should print on the IRIS screen, and then the screen should turn black and display axes. (**N.B.** If at any time the screen goes completely blank, touch any key on the IRIS keyboard. The IRIS has a dubious screen-saver "feature".)

Modes

The modes of use of the IRIS are as follows:

Interlisp-D display stream

In this mode, the IRIS is acting like a window on the Lisp machine. To see updates as they occur toggle the "Double Buffer" button on the IRISView Controller. It will respond "Single buffering". Then press the "2D-Home" button on the IRIS. To demonstrate this, try the use of commands like

```
(PROGN (DSPCOLOR 'BLUE \IRISSTREAM)
       (PRINTOUT \IRISSTREAM T "Let's here it for " )
     (DSPCOLOR  'RED \IRISSTREAM)
     (PRINTOUT  \IRISSTREAM  "COLOR." T)
     (DSPCOLOR  'BLUE \IRISSTREAM))
```

NOTE that the first time a font family is defined it will download the font after loading the spline font definitions. This will take a few minutes.

```
(DRAWLINE 0 0 400 200 2 NIL  \IRISSTREAM 'RED)

(FORCEOUTPUT IRISCONN)
```

Sinc e the SPP stream buffers, sometimes it will be necessary to (FORCEOUTPUT IRISCONN). (There is a function simply called F that will do this for you.)

These simple instructions indicate the use of the IRIS as an Interlisp Device Independent Graphics (DIG) stream. See the IRM for further examples common to all DIG devices. For interactive graphics, the following sections contain other uses of the IRIS.

Output stream for Sketch and TEdit

TEdit and Sketch can make use of the above display stream facilities. To see an example color document, do the following. (If you don't want to see the sketch in the document, you may omit loading sketch and sketchcolor, since they are slow to load.)

```
(FILESLOAD COLOROBJ SKETCH SKETCHCOLOR)

(TEDIT 'IRIS-EG.TEDIT)
```

To print a TEdit (or sketch) to the IRIS

First time: Press "2D-Home". In the title bar of the window, use the right button to bring up the default window menu. Pick the roll-off item Hardcopy. Rolloff into "hardcopy to printer", then choose "Iris". Choose "yes" to the menu saying "Make this the default printer".

Later times: The IRIS will now be the default printer, so just press "2D-Home"and choose Hardcopy from the default window.

To use Sketch

From the background menu, choose "SKetch". Build a closed polygon (with the entry in the sketch menu). Choose "change" in the sketch menu. Pick a control point of the polygon you just built. Pick "filling color" from the menu that appears. From the menu that appears, pick a filling color.

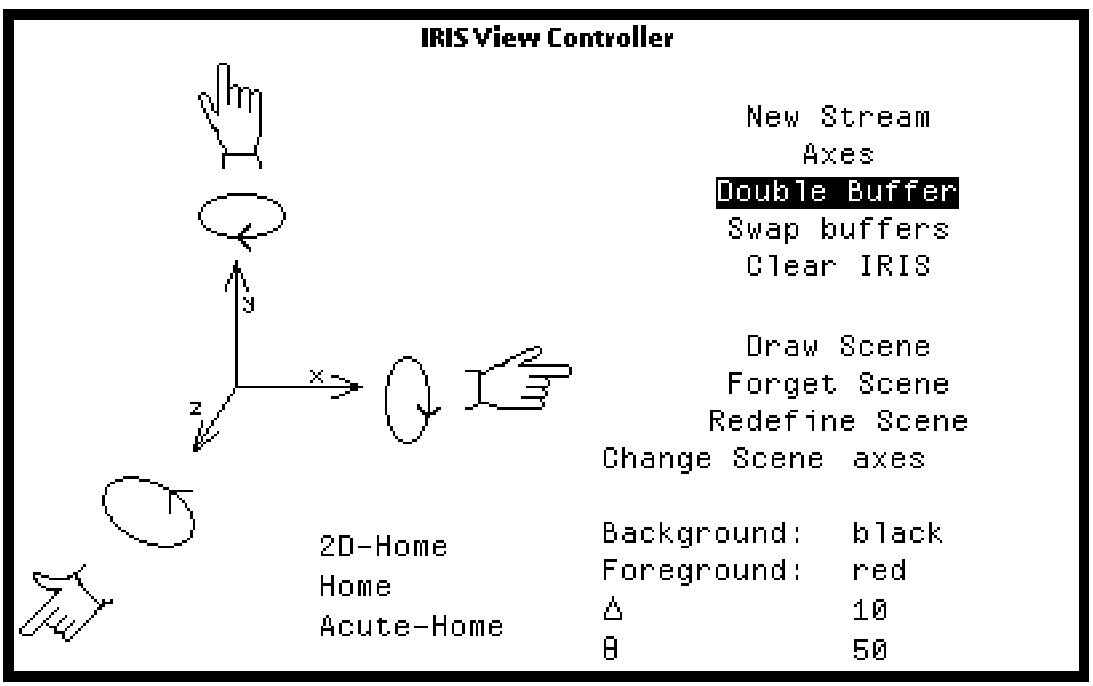
Clearing the IRIS

In the IRIS icon is an entry "clear IRIS". It will return the IRIS to the original state with the white screen, etc.

Defining new colors

At any time the color menu is presented, "new color" may be selected. RGB will provide color sliders and CNS will provide an english description of colors. A window will pop up asking for a name for the color. Type in a name, and that name will appear in the color menu from then on.

Using the IRISVIEW controller



N.B. When an IRISview controller is initially opened, the IRIS will be double buffering. This means that any drawing on the iris occurs on the back of two buffers. "Swap Buffers" must be chosen to bring the back buffer to the front. This includes "Clear IRIS" and other drawing commands. Double buffering can be turned off with the "Double Buffer" button, but moving images are much smoother with it turned on.

This window allows you to define objects on the IRIS and control your view of them.

Changing the scene:

First , select "change scene". You will be prompted with the predefined scenes as examples. Choose "axes". After a moment (during with "defining object for axes" is printed), an XYZ axes picture will appear on the screen. Try choosing each of the 3 home buttons to see the effect.

Press the "2d Home" button. (This actually means to move the view such that the IRIS simulates an 1108 screen in size and view). Now change the scene to "SKULL". Again after a delay, a Grateful Dead logo will be displayed.

Defining a scene:

We are going to define a scene that calls a function on the file IRISDEMOFNS so type

(FILESLOAD IRISDEMOFNS)

Button "Change Scene". Choose "New Scene" from the Menu. In response to "Scene Name", type "TETRA". In response to "Form to eval" click in the window the message is printed in and then type

(TETRA)   [carriage return]

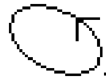
When the object is displayed, press each of the "Home" buttons to see different home positions. (2D-Home is perhaps best).

Viewing the scene:

Before viewing the scene, choose the "Double Buffer" option. Now hold the left button down on the item marked "Background:" in the view controller. Choose the background color for the object. White is often a good choice. (Note that sliding off the menu at this point will display "none" as the background. This means that each time the scene is drawn will be on top of the previous result. This can give very pleasing, flashy results. Experiment a bit. It’s great fun (and you are probably being payed for it!))

Adjusting the view:

Bug the part of the free menu marked ☞. This will move the image in the positive x axis. Similarly with the other hand icons. Holding down the shift key will reverse the direction in the same axis. Bugging the Δ entry in the view controller will let you change the "delta" for translation.

Now bug the part of the free menu marked . This will rotate the image in that axis. Holding down the shift key will reverse the direction of rotation.

Bugging the θ entry will let you choose a new "theta" for rotation (in degrees).

Home:

Bugging "home" will return the scene to the original view. It is often a good idea to choose "Home" before changing scenes.

Axes:

Bugging the "axes" entry in the menu will superimpose a set of three dimensional axes on the view for reference purposes. It defines the axes of rotation controlled by the view controller and the directions of translation. (The axes look best against a black background, I think)

Forgetting a Scene:

Reclaims the space inside the IRIS and removes scene names from the scene menu. All but one scene may be deleted.

Defining a scene containing a sketch (or Tedit):

Button "Change Scene". Choose "New Scene" from the Menu. In response to "Scene Name", type "IRIS-EG". In response to "Form to eval", click in the window the message is printed in and then type a carriage return.

The view controller will prompt with "Make object then type RETURN". Now, hardcopy the sketch exactly as above. (periodically the view controller will flash the screen to remind you that you are still making the object). When you are finished, type carriage return as before. The current scene name will be printed in the view controller.

Troubleshooting:

"Iris Terminall SPP not responding" printing in prompt window

This may occasionally print when the Dlion is too busy to service the SPP connection to the IRIS. If it repeats, however, this means that a previous IRIS stream has been lost. Bring up a process status window by choosing PSW in the background, then choose the process (Iris Terminal SPP#2 for instance) and bug "kill" in the PSW. Be sure to kill the correctly numbered process. "Iris Terminal SPP" is not the same as "Iris Terminal SPP#3".

Magic Variables

IRISNSHOSTNUMBER [Variable]

contains the 48 bit etherhostnumber of the IRIS. If you are etherbooting, this will be set automatically. Otherwise, set it to a string like "0#4000.12000.41504#0" as described on page 31.9 of the IRM.

\IRIS.DEBUG [Variable]

Defaults to NIL. If T, when fonts are created, only the first lowercase letters will be defined. This is much faster than loading the whole font.

IRISBOOTDIRECTORIES [Variable]

For users still running with the R1B version of the IRIS terminal program, the file concerned is called xiris. This is the file that should be copied to {core}irisbootfile. However, for users running R1C (also referred to as GL2), it is necessary to obtain an updated version of the terminal program from SGI. (It was named simply "iris" on the tape we received.) This variable must contain the list of directories where the boot files are to be found. (For instance, very fast booting may be obtained by copying the boot files to the {core} device and putting {core} at the front of IRIS.BOOTDIRECTORIES.)

IRIS.VERBOSE [Variable]

Defaults to T. Says whether or not status messages are printed during font creation etc.

\IRISSTREAM [Variable]

contains the IRIS stream that is current.

IRISCONN [Variable]

contains the SPP connection that is current.

KOTOLOGO

By: Masinter (Masinter.PA@Xerox.COM)

Uses: none

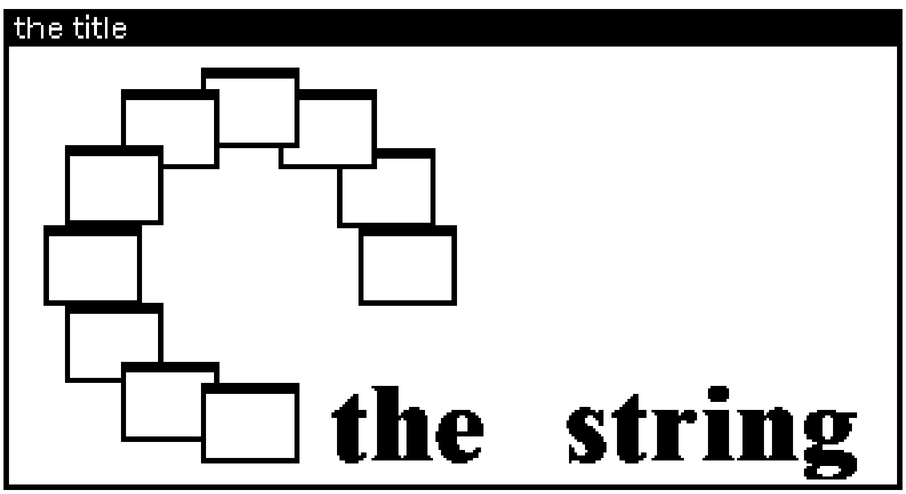
This document last edited on August 17, 1988.

INTRODUCTION

Makes a Koto-style logo window.

(KOTOLOGOW *string where title angledelta*) [Function]

Works like LOGOW did in Koto. Put *string* as the main logo name, with *title* in the window title. *angledelta* is the angle at which the little windows go. *where* is either a position or an old window. For example (KOTOLOGOW "the string" NIL "the title" 30) produces:



LispNerd

By: Maxwell (Maxwell.pa)

DICTCLIENT, DINFO

INTERNAL

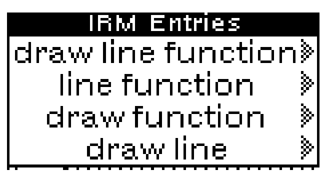
INTRODUCTION

The LispNerd provides a menu-based interface to the Interlisp Reference Manual. The data for the LispNerd is stored on the Dictionary Server, and is accessed via DICTCLIENT. Because of certain licensing agreements we have with Houghton-Miflin, the Dictionary Server should only be used by people within PARC. Hence LispNerd has not been released as a LispUsers package.

HOW TO USE THE LISPNERD

When you load the LISPNERD, it adds a new menu item called "Search IRM" to the background menu. Bugging "Search IRM" causes the LispNerd to prompt the user for keywords in the prompt window. It then produces a menu of items that have at least two of the keywords in their definitions (perhaps with submenus, if there is more than one class of items). Bugging one of these items will cause the LispNerd to fetch the documentation for that item using DInfo.

For example, if you type the input "draw line function", you will get the following menu:



where each of the entries has a sub-menu of the items that have the keywords listed in the entry.

If you type the input "date", you will get a menu all of the items in the Interlisp Reference Manual that have the word "date" in them. Putting parenthesis around items means that the items should all be treated as one keyword for the purposes of sorting them into groups. For instance, the input "(draw draws)(line lines) function" will look for all of the items that have "draw" OR "draws" in their definition.

LispNerd only fetches 50 entries at a time, so sometimes you will see an entry in the menu that says something like ". . . + 103 more". Clicking this item will cause the LispNerd to fetch the next 50 entries. Also, when there is more than one class of entries, sometimes a sub-menu will only list the number of entries it has, and not the entries themselves. To get the entries, click the menu item with the keywords in it, and LispNerd will recompute the menu with just those keywords.

Please send all bug reports to Maxwell.pa.

MICROTEK

By: Ron Clarke (RClarke.pa@Xerox.COM)

This document last edited October 31, 1988

INTRODUCTION

MICROTEK is an image processing software package that enables you to operate Microtek Models-300 and 300A Intelligent Image Scanners with the Xerox 1108 and 1186 workstations. The Microtek MS-300, 300A, and MSF-300C are high-resolution optical scanners that can convert text, artwork, photographs, etc, into digital form for processing by computer. The digitized images that are output to the computer contain up to 300 black and white dots for every linear inch of the original document. Page size can be as large as 8.5 by 14 inches. Sophisticated firmware in this scanner enables the user to set the scanning area and control brightness, contrast, scaling, shading and other characteristics of the scanned images through simple commands transmitted from the 1108 or 1186. Two basic scanning modes are supported: Line Art mode for accurate capture of completely black-and-white material, and Halftone mode for faithful reproduction of material with varied shading. Mixed-mode scanning is also available.

With the MICROTEK software package you will be able to: Set the scanner to capture images of all kinds, with desired visual effects, and transmit them to the 1108/1186, save scanned images to disk, floppy or file server for later reloading to recreate images and print scanned images to a Xerox 4045 or 8044 laser printer.

SOFTWARE REQUIRED

MICROTEK.DFASL

MICROTEKPRINT.DFASL (if you have a Xerox 4045 or 8044 laser printer)

DLRS232C.LCOM

EDITBITMAP.LCOM

READNUMBER.LCOM

4045XLPSTREAM.DFASL (if you have a Xerox 4045 laser printer)

FONTS USED

MODERN 10, 12 BOLD

Other useful software for manipulating the scanned image:

Lispuser's Packages:

ACTIVEREGIONS, ACTIVEREGIONS2, AIREGIONS, FILLREGION

HARDWARE REQUIRED

Xerox 1108 with RS232C port (E-30 upgrade kit). It is also recommended that the 1108 have 3.5 meg of memory and a floating point processor (CPE board) to enable faster scanning and creation of bitmaps.

Xerox 1186. It is also recommended that the 1186 have 3.7 meg of memory.

Microtek MS-300, MS-300A, or MSF-300C Intelligent Image Scanner with optional serial port.

CABLE CONFIGURATION

Note that the cable configuration is DIFFERENT for the MSF-300C scanner. Plugging a standard RS232C cable into the MSF-300C DB25 connector may result in damage to the equipment.

RS232C Port (DTE)		**MICROTEK MS-300, MS-300A - DB25 Connector**	
Signal	Pin	Pin	Signal
FGround	1	1	FGround
TD	2	3	RD
RD	3	2	TD
SGround	7	7	Sground

Pins 5, 6, 8 and 20 are jumpered together on the RS232C port end of the cable.

RS232C Port (DTE)		**MICROTEK MSF-300C- DB25 Connector**	
Signal	Pin	Pin	Signal
TD	21	3	RD
RD	9	2	TD
Ground	5,7	7	Ground

DOCUMENTATION REQUIRED

Microtek MS-300, MS-300A, or MSF-300C Intelligent Image Scanner Operation Manual

LOADING MICROTEK

Make sure that DIRECTORIES contains the directory where the required software is located. When the file MICROTEK.DFASL is loaded, the item "MicrotekScanner" will be added to the Background menu. If you have a Xerox 4045 or 8044 laser printer load MICROTEKPRINT.DFASL. If you have a Xerox 4045 laser printer load 4045XLPSTREAM.DFASL. Your 4045 laser printer should be connected to the TTY/DCE port.

RUNNING MICROTEK

The process of running the Microtek scanner software consists of three phases: Scanner initialization, scanning, and creating a bitmap of the scanned image that can eventually be printed. Each of these are controlled by different menus within the Microtek Scanner Control Window.

SCANNER INITIALIZATION

Set the Microtek scanner so that it is operating at 19200 baud by setting its internal DIP switches (See Microtek Operating Manual for details). Turn on the Microtek scanner. Select "MicrotekScanner" from the background menu and the Microtek Scanner Control window (figure 1) and Microtek Scanner Pagemap window (figure 2) will be created. (Note you may have do a control-E and retry if cursor flashes while trying to create the control window).The scanner pagemap window is used to select the area of the image to be scanned and to select the page length. The scanner control window is used to set all other scanner parameters, start and stop scanning as well as to initiate creation and printing of scanned image bitmaps. After these windows have been created, the RS232 port will be initialized to

19200 baud and an attention command will be sent to the scanner. If all cables are connected properly and the scanner is on, the message "MICROSCAN 300(A) V# is ready" will be displayed in the Microtek Status Window. If the cable is configured incorrectly or the scanner is not on or ready the messsage "Microtek Not Responding ... Check scanner and cable" will appear instead.

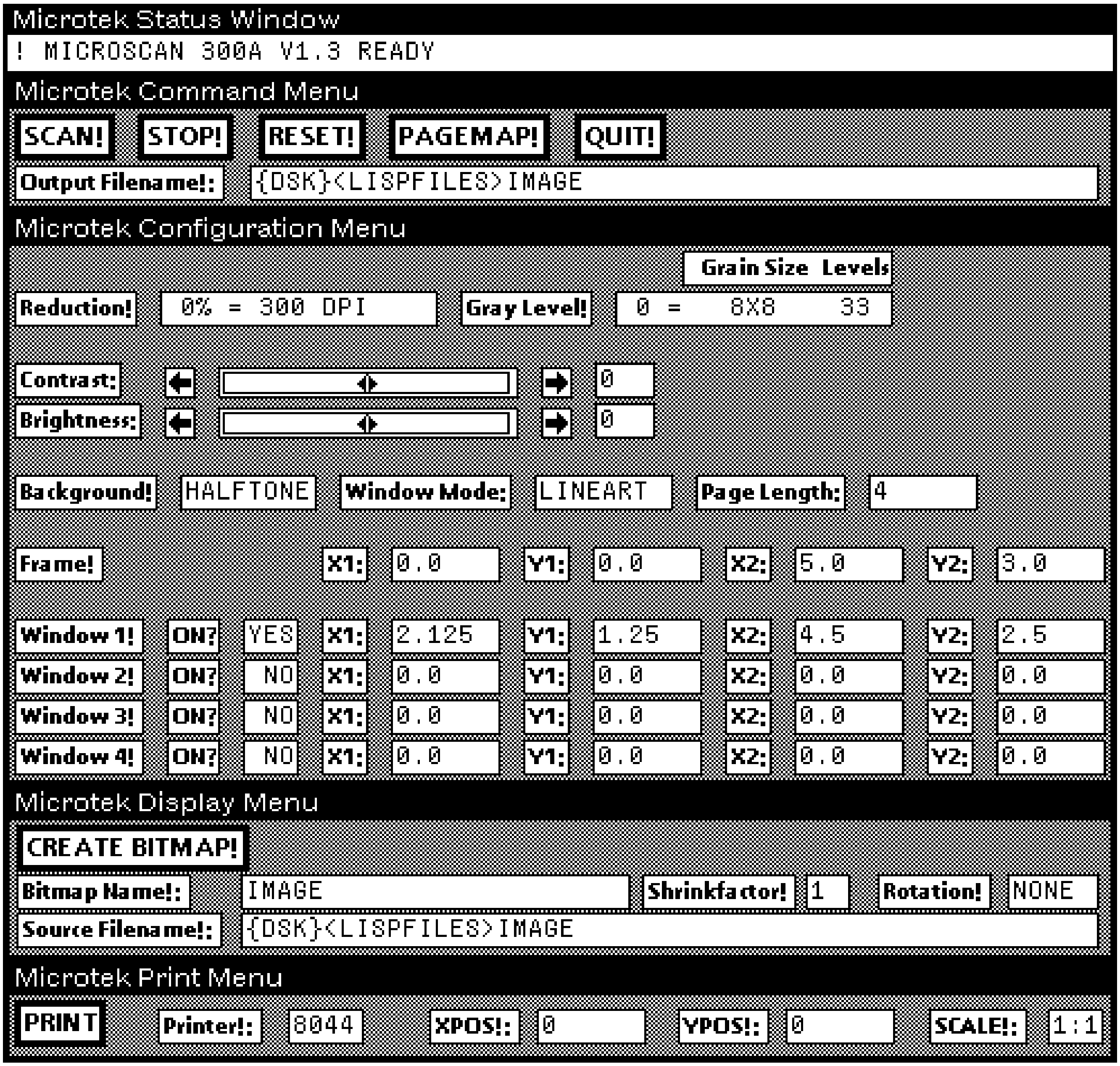


FIGURE 1 - MICROTEK SCANNER CONTROL WINDOW

Before scanning can be initiated, a number of parameters have to be set by the user via the Microtek Command Menu and Microtek Configuration Menu as follows:

Microtek Command Menu:

Output FileName Left buttoning on this item allows you enter the name of the file on disk, floppy or fileserver where the scanned data is to be saved. Be sure to type a carriage return to terminate this entry.

Microtek Configuration Menu:

Reduction Left button on the number next to the item Reduction and a menu will appear. Reduction can be changed from 0%, which corresponds to scanning at 300 dots per inch (DPI) to 75%, which corresponds to 75 DPI.

GrayLevel Left button on the number next to the item GrayLevel and a menu will appear allowing you to choose from a selection of gray levels based on grain size and number of gray levels within that grain size.

Contrast Left button on either the the left or right arrow to either decrease or increase the contrast setting.

Brightness Left button on either the the left or right arrow to either decrease or increase the brightness setting.

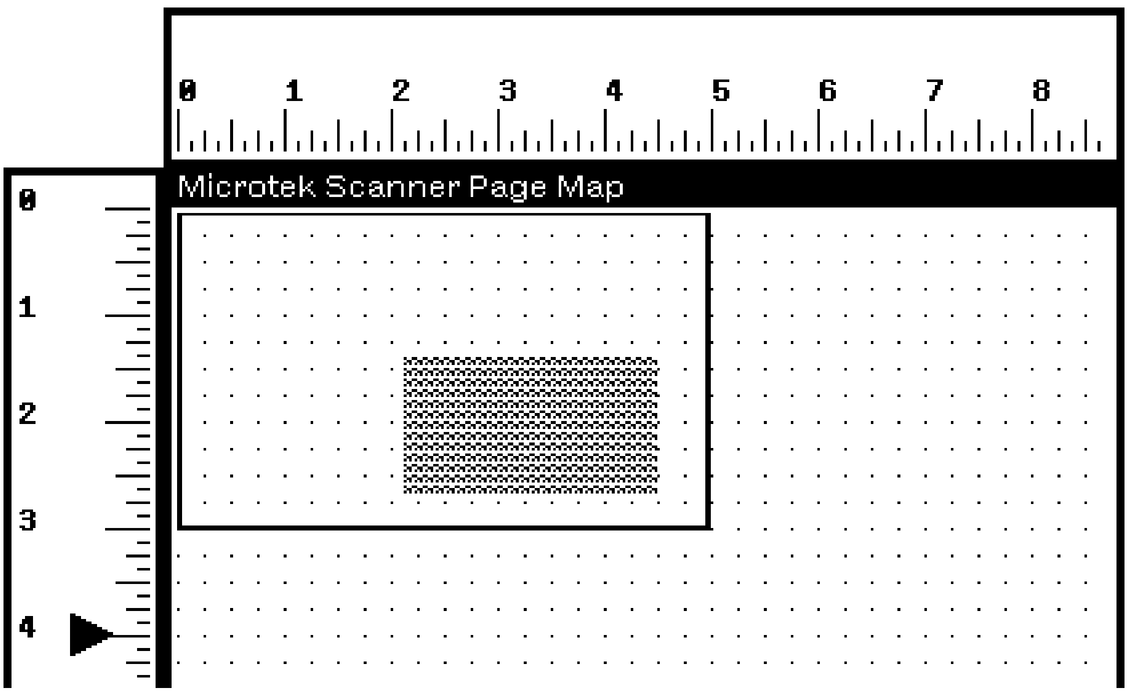


FIGURE 2 - MICROTEK SCANNER PAGEMAP WINDOW

BackGround Select either HALFTONE or LINEART as the primary scanning mode for the image. Line Art mode is for accurate capture of completely black-and-white material, and Halftone mode for faithful reproduction of material with varied shading.

Pagelength Move the cursor to the vertical ruler of the page map (figure 2). The cursor will change to a right pointing triangle. Position this triangle and left-button to select the pagelength. The page length will also show up in the configuration menu. The page length should be set so that it is longer than the actual page length of the document to be scanned. Otherwise you will get a paper jam message at the completion of scanning. The minimum page length is 3 inches and the maximum page length is 14 inches.

Frame The scanning frame is an area within the document that will be scanned. The maximum scanning frame is 8.5" by 14". Left button on the item Frame and you will be prompted to sweep out an area on the scanner page map to select the primary area to be scanned. The horizontal and vertical rulers and the page map grid dots can be used as a guide in determining the dimensions of the scanning frame. When the the scanning frame has been swept out, a box of the scanned area will be drawn on the page map and the actual X and Y coordinates of the top lefthand corner and lower righthand corner will appear next to corresponding items on the configuration menu (See Figure 2).

Windows 1- 4 Windows are areas within the scanning frame that are scanned in a different mode from the rest of the frame. If LINEART mode is selected as the background, all material in any windows you set will be scanned in Halftone mode, and vice versa.

The method used to set the windows is similar to that used to set the scanning frame except that you first need to specify whether the window is to be selected or not. This is done by left buttoning on the YES/NO indicator next to each window. A menu will pop-up and will allow you select "yes" or "no". After making your selection, left buttoning on the appropriate Window # will cause you to be prompted to sweep out an area within the scanning frame. Each selected window will be displayed and have a unique shade to it (See Figure 2). The only restriction is that the scanning mode must not change more than twice in one 8.5" horizontal scan line. Thus, if two windows lie across the same scan line they must extend to the edges of the page setting area. (Note that material to the left and right of the frame is scanned but not transmitted to the 1108.) You can select different windows for halftone vs lineart mode by switching between backgrounds. The item above WINDOW1 indicates which window mode is selected. An illegal window setting will result in an error message when you attempt to scan. Also note that the windows will be displayed on the scanner pagemap only if there is a "yes" next to the window number.

SCANNING

After the Microtek scanning parameters have been initialized, the document to be scanned should be placed in the scanner top-first with the image to be scanned facing away from the user. Scanning is initiated by left-buttoning SCAN on the Microtek Command Menu. The software first creates a scratch file in {CORE} for storage of the incoming data. It then sends the scanning parameters to the scanner and if all are valid the scanning process starts as indicated by movement of the rollers. You have up to 5 minutes to insert a document before the scanner automatically stops. After scanning has been completed you will be notified in the status window that it is saving the core file to the file specified in Output Filename. It takes approxiamtely 20 minutes to scan an 8.5" x 11" document at 300 DPI.

You may stop the scanning at any time by selecting STOP. The document will be ejected and the scanner reset. You can also explicitly reset the scanner by selecting RESET. This closes the scanner scratch file if it is open, sends a reset command to the scanner and then sends the attention command. If everything is reset properly, you will get the message "MICROSCAN 300(A) V# is ready" in the status window.

CREATING BITMAPS OF SCANNED IMAGES

The Microtek Display Menu is used to create bitmaps from a file that contains scanned data. Select SOURCE FILENAME and enter the name of the file that contains the scanned data. Select BITMAP NAME and enter the name of a variable that you would like the bitmap bound to. **Be sure to type a carraige return to terminate the entry of each of these items.** Left button on the number next to

SHRINKFACTOR and choose a factor by which you want the bitmap shrunk. The default value is 1. Left button on the item next to ROTATION and choose how you want the scanned image to be rotated. The default is "none." After these items have been set, you can then select CREATE BITMAP to start the bitmap creation process. The status window will be updated as it proceeds to create the bitmap and finally, you will be prompted to sweep out a scrollable window to display the bitmap. NOTE: Depending on the size of the bitmap, rotation may take a "very" long time and will look like your machine has frozen...be patient, it will come back. If you desire to save the bitmap(s) on a file you can do the following:

```
(SETQ filenameCOMS '((VARS bitmapname1 bitmapname2 etc))).
(MAKEFILE '{device}<directory>filename)
```

PRINTING BITMAPS OF SCANNED IMAGES TO A XEROX LASER PRINTER

If you have the package MICOTEKPRINT loaded you will have a MicrotekPrint Menu under your display menu (See Figure 1). Select BITMAPNAME on the display menu and enter the name of the bitmap that you would like to print. To select where on the page the bitmap is printed, left button XPOS and YPOS and enter a number. For the 4045 laser printer the values of XPOS can be between 0 - 2550 and YPOS, between 0 - 3300. 1" = 300 print units o 4045 . For an 8044 Interpress laser printer the values of XPOS can be between 0 - 21590 and YPOS, between 0 - 27940. 1" = 2540 Interpress units. The scale that an image is printed at is dependent upon its initial scanned reduction/DPI. You can increase or decrease the scale at which the bitmap is printed by buttoning on the number next to the item SCALE and selecting a scaling factor. On an 8044 Interpress printer a scale of 8:1 will magnify an image by 8 times on printing , 1:1 will print at true size and 1:8 reduce the image by 8 times. Values in between are also available. On a 4045 laser printer only a limited number of scale factor are availble and is dependent upon the original scan reduction as shown in the table below.

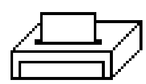
REDUCTION (%)	RESOLUTION (DPI)	SCALES ALLOWED
0	300	1:1, 2:1, 4:1
5	285	1:1, 2:1, 4:1
10	270	1:1, 2:1, 4:1
15	255	1:1, 2:1, 4:1
20	240	1:1, 2:1, 4:1
25	225	1:1, 2:1, 4:1
33	200	1:1, 2:1, 4:1
35	195	1:1, 2:1, 4:1

REDUCTION (%)	RESOLUTION (DPI)	SCALES ALLOWED
40	180	2:1, 1:1, 1:2
45	165	2:1, 1:1, 1:2
50	150	2:1, 1:1, 1:2
55	135	2:1, 1:1, 1:2
60	120	2:1, 1:1, 1:2
67	100	1:1, 1:2, 1:4
70	90	1:1, 1:2, 1:4
75	75	1:1, 1:2 , 1:4

Select PRINT to initiate the printing process. NOTE: The amount of reduction that you will be able to do is dependent upon the number bits that were originally scanned in. If you make the scale too small nothing will be printed out.

OTHER ITEMS AND GENERAL COMMENTS

On the Microtek Command Menu, left buttoning the item PAGEMAP will alternately open and close the scanner pagemap window. Left buttoning on the item QUIT will close the input and output streams to the scanner, shutdown the RS232C port and close the scanner pagemap and control windows. The following icon will be displayed if you shrink the Microtek Scanner Control window.



The Microtek Pagemap window will close when you shrink the Microtek Scanner Control window and has to be expicitely opened when the Microtek Scanner Control window is expanded again. This is done by buttoning on PAGEMAP in the Microtek Command Menu window.

Within Interlisp you normally cannot create bitmaps larger than approximately 2.1 million pixels (about 1400 x 1400). The Microtek scanner software allows you to create bitmaps much larger than this but at the cost of using a correspondingly large amount of virtual memory. If you are near your maximum vmemsize, as determined by comparing (VMEMSIZE) to (VOLUMESIZE 'volumename) , there is a good chance you could crash your system if you create a very large bitmap...caveat emptor. In addition you will not be able to call the function EDITBM to edit bitmaps larger than 2.1 million pixels

The reduction % used to scan the original image is stored on the property list of the atom that the bitmap is bound to. It is saved as the property "Resolution" and is in %. This is used to determine the appropriate values that will make an image 1:1 when printed. If you attempt to print a bitmap to an Interpress printer that was not created by use of the Microtek scanner software you will be prompted to enter a scale explicitely. The following table indicates the 8044 laser printer scale used for scanned images and can be used as a guide when attempting to print bitmaps not created by the Microtek software.

REDUCTION (%)	RESOLUTION (DPI)	SCALE
0	300	.240
5	285	.252
10	270	.266
15	255	.282
20	240	.300
25	225	.320
33	200	.360
35	195	.369
40	180	.400
45	165	.439
50	150	.480
55	135	.533
60	120	.600
67	100	.720
70	90	.800
75	75	.960

Further information about the Microtek scanner can be obtained from:

Microtek Lab Inc
16901 South Western Avenue
Gardena, California 90247
Tel: 213-321-2121, 800-654-4160

NSCOPYFILE

By: Bill van Melle (vanMelle@Xerox.com)

The module NSCOPYFILE modifies COPYFILE so that if both the source and destination files are on NS file servers, the copying is done by an NSFiling-specific copy routine. This routine copies *all* attributes of the source, including non-standard ones, such as those used by Viewpoint. Thus, you can safely copy Viewpoint files from inside Lisp without losing information. In addition, if the copy is from an NS file server to itself, the copy is performed by the server itself, which is considerably faster than shipping the file over the Ethernet.

In addition, you can also copy entire directories, by specifying directory names as the source and destination, e.g.,

```
(COPYFILE "{FS:}<Carstairs>Lisp>" "{FS:}<Calvin>Lisp>Current>")
```

The destination directory must not already exist, since this operation creates an entirely new directory, whose contents are a copy of all the source directory's offspring, to all levels. If the destination directory happens to exist but has no children, it is considered vestigial and is quietly deleted first (Lisp usually suppresses such directories when performing a directory enumeration).

You can also use RENAMEFILE in the same manner to either rename a directory, or to move an entire directory and its descendents to a new node in the file server's hierarchy, or to a new server altogether. You must, of course, have access rights to delete the source directory *and* all its children, and the destination must be on an NS file server.

A word about protection: when a file is copied or moved, the new file is given "defaulted" access rights, i.e., its protection is set as specified by its new parent (sub)directory, just as if you had created the file afresh by any other means. Thus, if the original file happened to have its own explicit protection, that protection is ignored. When copying or moving an entire directory, only the top-level directory receives default protection, so if any individual descendent file had non-default protection, that protection is copied verbatim. This can lead to confusion—you may want to use the NSPROTECTION module to change the new directory's descendents to default protection. See the documentation of NSPROTECTION for more discussion about protection issues. Note that if a file/directory is renamed within the same parent directory, the operation is considered merely "changing the name", and its protection is left unchanged.

Note: If you are using the FILEWATCH module, be aware that files being copied between NS servers do not appear (because the files are not opened by the normal Lisp file system).

NSPROTECTION

By: Bill van Melle (vanMelle@Xerox.com)

INTRODUCTION

The module NSPROTECTION provides a tool that enables you to easily change the protection of files and directories on Xerox NS file servers.

To install the module, load the file NSPROTECTION.LCOM. Also, Your NS file server must be running Services release 10.0 or later.

THE PROTECTION MECHANISM

An NS File Server maintains a protection for each file and (sub)directory on the server. In most cases, the protection is not specified explicitly, but rather is inherited from a file's parent directory, making it easy to maintain consistent protection over an entire branch of the file system hierarchy.

The protection is specified as a set of pairs <access rights, name>. The name can be the name of an individual user or a group. The name can also be a pattern of the restricted form *:domain:organization, *:*:organization, or *:*:*. The access rights granted to any particular user are the most general of those in the pairs that match the user's name (by exact match, pattern or membership).

The following five kinds of access rights are independently specified (the term "file" here can also denote a directory in the places where that makes sense):

Read — The user may read the file's content and attributes. In the case of a directory, the user may enumerate files in it.

Write — The user may change the file's content and attributes, and may delete the file. In the case of a directory, the user may change the protection of any of the directory's immediate children.

Add — (Applies only to directories) The user may create files in the directory (i.e., add children).

Delete — (Applies only to directories) The user may delete files from the directory (i.e., remove children).

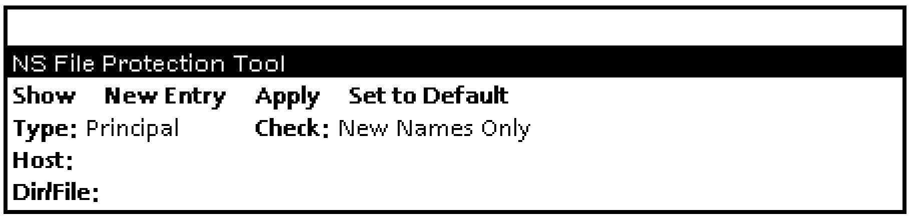
Owner — The user may change the file's access list.

In the case of directories, it is also possible to independently specify the directory's own protection and the protection that its children inherit by default. In most cases, the latter simply defaults to the former, and it is usually best to keep it that way for simplicity. However, there might conceivably be cases where, for example, you would want a user to be able to read the files in a directory, but not be able to enumerate it, or vice-versa.

Note that there can be problems when giving a more lenient protection to a file or directory than to its parents, depending on what software is going to be used to gain access to the file. For example, if your default directory protection grants access only to you, and you want to allow a user to read a particular file stored in your directory, then you can change the protection on just that file to allow Read access. However, the user will have to know the exact name of the file in order to read it, since she won't be able to enumerate the directory to search for the file. Specifying the exact file name works fine from Lisp, but other software that gets to a file by starting at the top and working its way down through the hierarchy would be unable to get to the file.

USER INTERFACE

To use the tool, select "NS Protection" from the background menu (if your menu has a "System" item, it's a subitem underneath it), or call the function (NSPROTECTION). You are prompted for a place to position the tool's window. Be sure to leave space below the window for the protection information that will follow.



The tool window has four command buttons across the top, two switches labeled **Type** and **Check**, and two fill-in fields for the host and file name. Holding a mouse button down over any of these items for a couple of seconds will display a help message in the prompt window.

To view or change the protection of a file or directory, first fill in the **Host** and **Dir/File** fields. You can edit these fields by clicking with the mouse anywhere inside the existing text (if any), or by clicking with the LEFT button on the boldface label. If you click with RIGHT on the label, then any existing text is first erased. Typing the Next or Return key moves to the next field. [See the FreeMenu documentation for more information about text editing.]

You can either enter the host and directory separately, e.g.,

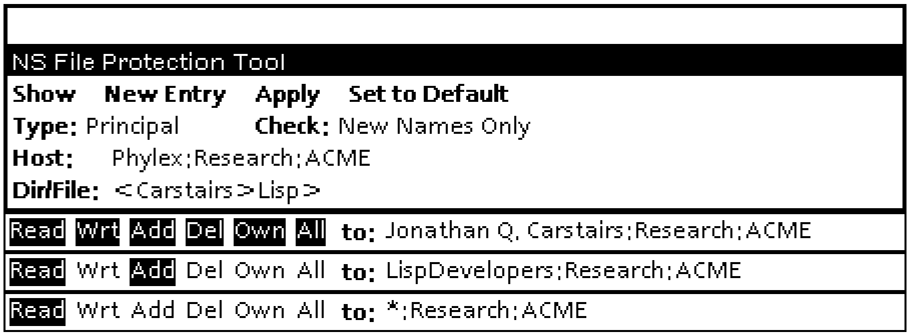
Host: Phylex
Dir/File: Carstairs>Lisp

or enter a file name in the usual Lisp syntax in the **Dir/File** field, e.g.,

Host:
Dir/File: {Phylex:}<Carstairs>Lisp>

This latter form is intended to make it easy to copy-select the name of the directory or file from another source, such as a FileBrowser window; the host in the full name overrides any name in the **Host** line.

To see the protection of a file or directory, click on the command **Show**. The protection is displayed as a series of editable one-line windows beneath the main window. In each line is a set of access rights and a Clearinghouse name or pattern to which those rights are granted; for example,



The highlighted buttons indicate which of the five access rights (Read, Write, Add, Delete, Owner) are granted to the name on the right. If the displayed protection was inherited from its parent subdirectory, rather than having been explicitly set, this fact is noted in the prompt window.

To change the protection of a file or directory, set up the protection entries as desired, then click on the command **Apply**. The usual procedure is to use the **Show** command to see the current protection, then edit one or more entries. Clicking on one of the first five buttons toggles it; clicking on **All** either sets all five (if **All** was previously unhighlighted) or clears all five. In addition, setting either **Write** or **Add** also sets **Read**, since they are of little use without read access (you can, however, clear **Read** if you really meant it). The name following **to** is edited in the same manner as the **Host** and **Dir/File** items above. As with most other places in the system, the name you type can omit the domain and organization, in which case the tool will fill in the local defaults; you can also use nicknames, which will be replaced by the Clearinghouse full names (assuming checking is on).

To add an additional entry, click on the command **New Entry**. This adds a new line to the existing set of protection entries, which you can edit as appropriate. To remove a set of access rights completely for an existing name, either clear all five access buttons (most easily done by clicking once or twice on **All**), or clear the name from the **to** field (by clicking on it with the RIGHT mouse button). Any such cleared lines will be removed by the Apply command.

You can also change the protection of a file back to "default" by clicking on the command **Set to Default**. Following this command, the protection of the specified file is inherited from its parent directory. This is usually the best way to "undo" a changed protection, because then any changes to the protection of its parent, or parent's parent, etc., will have the expected effect on all its children.

For the **Apply** and **Set to Default** commands, you may also specify a group of files, rather than a single file, by giving a file pattern—a name with asterisks serving as wild cards to match zero or more characters. Any pattern acceptable to the File Browser can be used. The tool enumerates the specified set of files and applies the specified protection to each. The enumeration is made to all levels (infinite depth), so affects files both in the immediate directory and also in its subdirectories, and subdirectories of those, etc. The enumeration does not, however, include the top-level subdirectory itself; e.g., "<Carstairs>Lisp>*" matches all files (including subdirectories) anywhere in the directory <Carstairs>Lisp>, but does not include <Carstairs>Lisp> itself.

Note that applying a protection to a directory is different from applying the same protection to the files in it, because of defaulting. If you apply a protection to <Carstairs>Lisp>*, it changes the protection of every file currently in the directory, but any new files added after the change still inherit the protection of the directory <Carstairs>Lisp>. On the other hand, applying a protection to the directory <Carstairs>Lisp> itself affects all current and future files in the directory, *except* any files that already have an explicit protection currently set. To reduce confusion, it is thus preferable to apply protections to subdirectories, rather than individual files, if you want to control a whole group of files. If you have a subdirectory containing files of miscellaneous protection that you would like to make uniform, the best procedure is to set the desired protection on the subdirectory itself, and then use the **Set to Default** command with a pattern (e.g., <Carstairs>Lisp>*) to reset all the individual files to defaulted.

The **Apply** command looks up in the Clearinghouse each of the names in the individual protection entries to make sure that they are valid, and replaces aliases (nicknames) with the canonical names. It then tells the file server to change the protection as indicated. The extent to which the **Apply** command checks names is controlled by the **Check** item in the second line of the tool window. It has four possible settings:

New Names Only — This is the default setting. The tool checks any names that you have entered or changed, but assumes that names returned by the Show command were correct.

All Names — The tool checks all names, regardless of source. You might want to do this to convert an existing protection entry into canonical form, or check that all the names are still valid.

Never — The tool never checks names; it assumes you meant exactly what you typed. You might want this setting, for example, if one of the names you are entering is registered only in a distant Clearinghouse not currently accessible.

I really mean it — Not only does the tool not check the names, it also doesn't balk if you tell it to take certain unlikely actions, such as changing a top-level directory to default protection, setting a completely null protection, or setting a protection in which nobody has Owner rights (which means the protection can only be changed by someone with Write access to the parent, if any). This setting is "one-shot"—it reverts to "New Names Only" after you issue the next command.

The **Type** item in the second line of the tool window controls which of a directory's two protection attributes is displayed or set. The initial setting is "Principal" and is the one that should normally be used (it coincides with the Lisp file attribute PROTECTION, or "Access List" in NS Filing parlance). The other setting is "Children Only". When the protection type is set this way, the tool deals with the protection that is inherited by default by the directory's children, the attribute called "Default Access List" in NS Filing parlance. Ordinarily, this attribute is defaulted, in which case the directory's principal protection is also used as its children's default protection. Using the **Apply** command changes the Default Access List to the value you specify; using the **Set to Default** command changes it back to defaulted. The **Show** command displays the directory's Default Access List if it has one; otherwise, it displays the principal protection and notes this fact in the prompt window.

The **Type** item is irrelevant for non-directory files (and, in fact, the tool sets it back to "Principal" if it has been changed). When the file is a pattern, the tool always sets the Principal protection; in the case of any subdirectories matching the pattern, it sets the Principal protection to that specified in the window and the Default Access List to "default".

As an additional convenience feature, when you request to **Show** the "Principal" protection of a top-level directory, the tool also displays in the prompt window the directory's current page usage and allocation.

POSTSCRIPT

By: Matt Heffron (BEC.HEFFRON@ECLA.USC.EDU)

INTRODUCTION

The PostScript package defines a set of imageops for printers which understand the PostScript page description language by Adobe. At Beckman we have successfully used TEdit, Sketch, LISTFILES, and HARDCOPYW to an Apple LaserWriter and an AST TurboLaser PS. The PostScript imagestream driver installs itself when it is loaded. All symbols in the PostScript driver are located in the INTERLISP: package.

VARIABLES

POSTSCRIPT.FONT.ALIST [InitVariable]

POSTSCRIPT.FONT.ALIST is an ALIST mapping Xerox Lisp font names into the root names of PostScript font files. It is also used for font family coercions. The default value should be acceptable for any of the fonts which are built into the Apple Laserwriter.

POSTSCRIPTFONTDIRECTORIES [InitVariable]

POSTSCRIPTFONTDIRECTORIES is the list of directories where the PostScript .PSCFONT font files can be found. The default value is: ("{DSK}<LISPFILES>FONTS>PSC>").

\POSTSCRIPT.SHORTEDGE.SHIFT [InitVariable]

\POSTSCRIPT.SHORTEDGE.SHIFT is the distance (in points) to shift the image perpendicular to the short edge of the paper. A positive value gives a shift upward in portrait mode, and to the right in landscape mode. The default value is: 0.

\POSTSCRIPT.LONGEDGE.SHIFT [InitVariable]

\POSTSCRIPT.LONGEDGE.SHIFT is the corresponding variable for shifts perpendicular to the long edge of the paper. A positive value here gives a shift to the right in portrait mode and downward in landscape mode. The default value is: 0.

\POSTSCRIPT.SHORTEDGE.PTS [InitVariable]

\POSTSCRIPT.SHORTEDGE.PTS indicates the printable region of the page (in points) along the short edge of the paper. It should be adjusted to allow for any shifts of the image (see above). The default value is: 576 (= 8 inches).

\POSTSCRIPT.LONGEDGE.PTS [InitVariable]

\POSTSCRIPT.LONGEDGE.PTS indicates the printable region of the page (in points) along the long edge of the paper. It should be adjusted to allow for any shifts of the image (see above). The default value is: 786.24 (= 10.92 inches).

HINT

> The AST TurboLaser PS has an imageable area on the page which is a different size than that of the Apple LaserWriter. The values of \POSTSCRIPT.SHORTEDGE.PTS and \POSTSCRIPT.LONGEDGE.PTS for the AST are 575.76 and 767.76, respectively.

\POSTSCRIPT.MAX.WILD.FONTSIZE [InitVariable]

\POSTSCRIPT.MAX.WILD.FONTSIZE indicates the maximum point size that should be returned from FONTSAVAILABLE when the SIZE argument is wild (i.e. *). All integer pointsizes from 1 to \POSTSCRIPT.MAX.WILD.FONTSIZE will be indicated as available. The default value is: 72.

POSTSCRIPT.PREFER.LANDSCAPE [InitVariable]

POSTSCRIPT.PREFER.LANDSCAPE indicates if the OPENIMAGESTREAM method should default the orientation of output files to LANDSCAPE. It can have one of three values: NIL, T, or ASK. NIL means prefer portrait orientation output, T means prefer landscape, and ASK says to bring up a menu to ask the preferred orientation if it wasn’t explicitly indicated in the OPENIMAGESTREAM call (with the ROTATION option). The default value is: NIL.

POSTSCRIPT.TEXTFILE.LANDSCAPE [InitVariable]

POSTSCRIPT.TEXTFILE.LANDSCAPE indicates if the printing of TEXT files (e.g. LISTFILES, ...) should force the orientation of output files to LANDSCAPE. The default value is: NIL.

POSTSCRIPT.BITMAP.SCALE [InitVariable]

POSTSCRIPT.BITMAP.SCALE specifies an independent scale factor for display of bitmap images (e.g. window hardcopies). Values less than 1 will reduce the image size. (I.e. a value of 0.5 will give a half size bitmap image.) The position of the scaled bitmap will still have the SAME lower-left corner (i.e. the scaled bitmap is not centered in the region of the full size bitmap image). The default value is: 1.

HINT

> Setting POSTSCRIPT.BITMAP.SCALE to 0.96, instead of 1, will give cleaner BITMAP images on a 300 dpi printer. (This corrects for the 72 ppi imagestream *vs*. the 75 dpi printer, using 4x4 device dots per bitmap pixel.) Also, values of 0.24, 0.48 and 0.72, instead of 0.25, 0.5 and 0.75, will also give cleaner images for reduced size output. In general, use integer multiples of 0.24 for a 300 dpi printer.

POSTSCRIPT.TEXTURE.SCALE [InitVariable]

POSTSCRIPT.TEXTURE.SCALE specifies an independent scale for the display of bitmap textures. The value represents the number of device space units per texture unit (bitmap bit). The default value is 4, which represents each bit of the texture as a 4x4 block, so that textures are approximately the same resolution as on the screen (for 300 dpi output devices, such as the Apple Laserwriter).

The PostScript package extends the allowed representations of a texture, beyond 16-bit FIXP and 16x16 bitmap, to ANY square bitmap. (If the bitmap is not square, its longer edge is truncated from the top or right to make it square.) Use this feature with caution, as large bitmap textures, or sizes other than multiples of 16 bits square, require large amounts of storage in the PostScript interpreter (in the printer controller), and can cause limitcheck errors when actually printing.

Anywhere that a texture or color can be used on an imagestream or in the specification of a BRUSH, you can instead give a FLOATP between 0.0 and 1.0 (inclusive) to represent a PostScript halftone gray shade. (0.0 is black and 1.0 is white. Specifically, the value sets the brightness of the shade.) The value you specify will not be range checked, and will be passed directly through to the PostScript setgray operator. (E.g. you can pass 0.33 as the color to DRAWLINE to get a dark gray line with approximately 67% of the pixels in the line black.)

POSTSCRIPT.IMAGESIZEFACTOR [InitVariable]

POSTSCRIPT.IMAGESIZEFACTOR specifies an independent factor to change the overall size of the printed image. This re-sizing affects the entire printed output (specifically, it superimposes its effects upon those of POSTSCRIPT.BITMAP.SCALE and POSTSCRIPT.TEXTURE.SCALE). Values greater than 1 enlarge the printed image, and values less than 1 reduce it. An invalid POSTSCRIPT.IMAGESIZEFACTOR (i.e. not a positive, non-zero number) will use a value of 1. The BITMAPSCALE function for the POSTSCRIPT printer type does NOT consider the POSTSCRIPT.IMAGESIZEFACTOR when determining the scale factor for a bitmap.

MISCELLANEOUS

The SCALE of a PostScript imagestream is 100. This is to allow enough resolution in the width information for fonts to enable TEdit to correctly fill and justify text.

The first time any PostScript imagestream is created (even if only to hardcopy a bitmap or window) the DEFAULTFONT is instantiated (unless a FONTS option was given to the OPENIMAGESTREAM, in which case the initial font for the imagestream will be set to that font, or to the CAR if a list).

The PostScript imagestream method for FILLPOLYGON uses the global variable FILL.WRULE as the default value for the WINDINGNUMBER argument. (This is the same variable which is used by the DISPLAY imagestream method for FILLPOLYGON.)

The PostScript imagestream method for OPENIMAGESTREAM (and, therefore, SEND.FILE.TO.PRINTER), supports an IMAGESIZEFACTOR option to change the size of the printed image. The IMAGESIZEFACTOR re-sizing is combined with the POSTSCRIPT.IMAGESIZEFACTOR to produce an overall re-sizing of the printed image. A HEADING option is also supported to give a running header on each page of output. The value of the HEADING option is printed at the top left of the page, followed by "Page " and the appropriate page number. They are printed in the

DEFAULTFONT (unless a FONTS option was given to the OPENIMAGESTREAM, in which case it will be that font, or to the CAR if a list).

The PostScript package is contained in the files: POSTSCRIPT.LCOM & PS-SEND.LCOM, with the source in the files: POSTSCRIPT & PS-SEND. The module PS-SEND.LCOM is required and will be loaded automatically when POSTSCRIPT.LCOM is loaded. It contains the function which is called by SEND.FILE.TO.PRINTER to actually transmit the file to the printer. It is, by its nature, quite site specific, so it is in a separate file to make modifying it for any site relatively simple. System record declarations required to compile POSTSCRIPT can be found in EXPORTS.ALL.

I'm pretty sure that the output generated by the PostScript imageops fully conforms to the Adobe Systems Document Structuring Conventions, Version 2.0, January 31, 1987.

Including Other PostScript Operations

If you wish to insert your own specific PostScript operations into a PostScript imagestream, you can do so with the following functions:

(POSTSCRIPT.OUTSTR *STREAM STRING*) [Function]

POSTSCRIPT.OUTSTR outputs a string or value to the imagestream. *STREAM* must be an open PostScript imagestream. *STRING* is the value to output (STRINGP and LITATOM are most efficient, but any value can be output (its PRIN1 pname is used)).

(POSTSCRIPT.PUTCOMMAND *STREAM STRING*$_1$ *... STRING*$_n$) [NoSpread Function]

POSTSCRIPT.PUTCOMMAND is more general for sequences of commands and values. It calls POSTSCRIPT.OUTSTR repeatedly to output each of the *STRING*$_i$ arguments to *STREAM*.

(\POSTSCRIPT.OUTCHARFN *STREAM CHAR*) [Function]

\POSTSCRIPT.OUTCHARFN is used to output the characters forming the text of a PostScript string (e.g. the argument to a show or charpath operator). *STREAM* is the open PostScript imagestream to output to, and *CHAR* is the CHARCODE of the character to output. The ***/*** (slash), **(** and **)** (parenthesis) characters will be quoted with ***/***, and characters with ASCII values less than 32 (space) or greater than 126 (tilde) will be output as **/nnn** (in octal). \POSTSCRIPT.OUTCHARFN will output the **(** character to open the string, if necessary. Use POSTSCRIPT.CLOSESTRING (below) to close the string.

(POSTSCRIPT.CLOSESTRING *STREAM*) [Function]

POSTSCRIPT.CLOSESTRING closes a PostScript string (e.g. the argument to a show or charpath operator). *STREAM* is the open PostScript imagestream. It is important to use POSTSCRIPT.CLOSESTRING to output the **)** character to close the string, because it also clears the stream state flag that indicates that a string is in progress (otherwise, the next POSTSCRIPT.PUTCOMMAND would output the commands to close the string and show it).

Warning

Do not attempt to create a PostScript font larger than about 600 points, as much of Interlisp's font information is stored in SMALLP integers, and too large a font would overflow the font's height, or the width for any of the wider characters. (I know that 600 points is a ridiculously large limit (about 8.3 inches), but I thought I'd better mention it, or someone might try it!)

Changes from the Lyric Release

The Medley release of this PostScript imagestream driver changed the default value of POSTSCRIPT.TEXTFILE.LANDSCAPE from T to NIL. It also added the support for the HEADING option.

Known Problems/Limitations

The output generated for a PostScript imagestream is rather brute force. It isn't particularly careful to generate the smallest output file for a given sequence of operations. Specifically, it often generates extra end-of-lines between PostScript operator sequences (this has no effect on the printed output, only on the file size).

Using BITMAPs or Functions as BRUSH arguments to the curve drawing functions is not supported, nor is using a non-ROUND BRUSH with DRAWCIRCLE or DRAWELLIPSE.

There is no support for NS character sets other than 0, and there is no translation of the character code values from NS encoding to PostScript encoding.

There is no support for color.

\POSTSCRIPT.OUTCHARFN is pretty wimpy in its handling of TAB characters. It just outputs 8 SPACEs for the TAB.

I haven't yet documented how to build the .PSCFONT files for any new fonts that become available, I'll do that eventually.

PS-PATCH

By: Will Snow (snow.envos@xerox.com)

Requires: POSTSCRIPT

This module fixes some bugs in software other than postscript. If you are going to load sketch, load it BEFORE you load this patch. If you load sketch after loading this patch, evaluate the following form in an Interlisp Executive:

(MOVD (QUOTE NEW-SK-PICK-FONT) (QUOTE SK.PICK.FONT) NIL T)

This will make the printing of sketches with text in them reasonable.

PS-RS232

By: Matt Heffron (mheffron@orion.cf.uci.edu)

Requires: POSTSCRIPTSTREAM, PS-SEND, DLRS232C

The module PS-RS232 defines a printing host named PS-RS232 which sends PostScript output to a printer over the {RS232} port of the 11xx. It also puts a function onto AROUNDEXITFNS which reinitializes the {RS232} port after returning from LOGOUT. The BaudRate and other parameters of the {RS232} port are controlled by the following variables.

VARIABLES

PS-RS232-BAUD [InitVariable]

This is the BaudRate for the {RS232} port output stream. Defaults to: 9600.

PS-RS232-DATABITS [InitVariable]

This is the BitsPerSerialChar for the {RS232} port output stream. Defaults to: 8.

PS-RS232-PARITY [InitVariable]

This is the Parity for the {RS232} port output stream. Defaults to: NONE.

PS-RS232-STOPBITS [InitVariable]

This is the NoOfStopBits for the {RS232} port output stream. Defaults to: 1.

PS-RS232-FLOWCONTROL [InitVariable]

This is the FlowControl for the {RS232} port output stream. Defaults to: XOnXOff.

UPCSTATS

By: Larry Masinter (Masinter.pa@Xerox.com)

INTERNAL

This document last edited on 11 October 84

UPCSTATS is for gathering statistics about where Dorado microcode is spending its time. (It only works on Dorados.) It samples the microcode's PC while running something, and then plots it in a histogram. It really doesn't help much unless you are familiar with the organization of the Dorado Interlisp-D microcode, and want to analyze it.

(UPCSTATS *form dolistflg*) [Function]

will EVAL *form* while gathering statistics, and then print out a histogram. If *dolistflg* is NIL, the output will go to the current output file (NIL). The first time you run UPCSTATS, it will ask you for the name of a ".MB" file. This is a Dorado Microcode Binary, and you need to get the version of "DoradoLisp.MB" that corresponds to the "DoradoLispMC.EB" that is on your local disk. Normally this is on the "Basics" release subdirectory.

Once you've done a UPCSTATS, you can print the output again, merely by calling (PLOTPCS).

SAMPLE OUTPUT:

```
Microcode PC Sample:  Each * = 11 count, or      .03%
AEMUNOTREADY                                                    |***************
(.5154639)
NOSKIP                                                                        |*
(.5500377)
EFFADRPCREL+1      |*
                                             +2                               |*
(.6254715)
LDA3                                                                          |*
(.6569022)
LDA23              |**
                          +1                                                  |*
(.7669097)
LDAIX                                                                      |****
(.9146342)
LDAX                                                                       |****
(1.059215)
STACKGETSMD                                                        |************
(1.480387)
STA2               |*
```

The following notes explain my (woz) idea of what the edit-interface should be responsible for, and what constitutes and "editor".

An editor is a symbol who's function takes the args (structure props options) and starts an interactive editor on the structure. PROPS is a property list, and OPTIONS is a list of keywords, both affecting the behavior of the editor. The property :completion-fn specifies the function to be called when the user completes the edit. The completion-fn will be called with the arguments (structure props options changed?), where structure is the edited structure (even on abort, so that the edit interface could implement "undo abort"), props and options are as specified on the call to the editor and will be used to figure out what to do with this completion, and changed? is NIL (no changes made), T (changes made), or :ABORT (user wants to abort changes). It is then up to the completion function to do the right thing with the result of the edit.

The goal is that the editor doesn't know anything about who started it, where the structure came from, or what to do with it when it's done. And the edit-interface doesn't know anything about the editor's data structures.

In the case of open edit sessions (open or shrunk):
If the editor is told to start an edit, the editor must look for one already existing that matches (this can't be the responsibility of the edit interface, because it doesn't know about existence of open edits). the editor should restart the existing edit, processing any new props or options appropriately.

The markaschanged issue:
Since the editor knows it may have open edits, it needs to provide a hook for when the world gets changed underneath the editor. in this world this means markaschanged. in this case the editor should try to restart itself with the new structure. in other words, sedit::markaschangedfn should call edit-definition to start a new edit. the editor will then notice it has a matching edit open and restart itself with the new info.
This model is complicated by the fact that markaschanged gets called as a result of completion. Presently *ingore-changes-on-completion* controls the behavior in this case. Ideally, the editor would just say "i know it got changed, i just changed it!", and it would ignore the call.

The new version of SEdit (1/25/91) is very close to this definition, with the following exceptions, which can be fixed upon implementation of this edit inteface design:
- the completion-fn is called with (context structure changed?) since all of SEDITE's completion-fns expect these args. this should be fixed in handle-completion.
- in the abort case, undo is run until there are no more changes to undo, since sedit is sometimes handed structures "in place", destructive edits need to be undone, and thus the completion-fn never gets to see the edits. this can be fixed in the function complete.
-

To make this editor active, call (il:editmode editor-name), where editor-name is the symbol defined above. The function xcl::edit will then start the active editor.

xcl::edit-expression provides an example of starting the editor on an unnamed structure, where eqness is expected upon completion.

xcl::edit-definition provides a replacement for il:editdef, if il:getdef and il:putdef work correctly on all types.

:dontwait
  the editor (sedit) should not wait. don't wait is part of the editinterface features. if its going to wait, the editinterface should create the event and bind it in the completionfn to the notified by the completionfn.

need published edit-expression command. same as ed, but takes and expression, waits for once-only completion, and returns the expression. eg for FIX

sedit should provide a published interface to creating gaps, for use by make-prototype-defn.

edit interface may need a way to ask editor if it has an open edit for name/type

deal with editdates

an editor used to install itelf by replacing EDITL and EDITE, then editmode was created to look for Definition-for-EDITE property on the symbol returned by editmode. now an editor is defined as a function which takes (expr props options) and implements at least the props :completion-fn and :root-changed-fn. EDITMODE then returns the name of this function so the current editor can be applied.

Notes on making a Sun Loadup

update Jan. 25, 1990 by osamu

In a medley sysout on cottonmouth do the following:

```
;;CONN {DSK}/cottonmouth/users/medley/sources/
```

-- Make sure all the files are current. There are SUN specific changes to
-- the following files:

-- FILESETS: took PUP and LEAF out of 1LISPSET

```
(CL:IN-PACKAGE "IL")
```

-- make copyfiles go faster

```
(SETQ COPYFILESENUMERATE NIL)

(COPYFILES '{ERIS}<LISPCORE>SOURCES>*.*
           '{DSK}/cottonmouth/USERS/MEDLEY/SOURCES/ '(>A))

(COPYFILE '{ERIS}<LISPCORE>SUNLOADUP>FILESETS 'FILESETS)

(COPYFILE '{ERIS}<LISPCORE>SUNLOADUP>LOADUP.LISP 'LOADUP.LISP)

(COPYFILE '{ERIS}<LISPCORE>SUNLOADUP>FIX-ETHER.LCOM 'FIX-ETHER.LCOM)

(COPYFILES '{ERIS}<LISPCORE>SUNLOADUP>MAIKOLOADUPFNS.*
'{DSK}/cottonmouth/USERS/MEDLEY/SOURCES/ '(>A))
```

-- You will need the instructions on your local directory.

```
;;;(COPYFILE '{ERIS}<LISPCORE>SUNLOADUP>HOWTO-LOADUPSUN.TXT '{DSK}HOWTO-
LOADUP-SUNLISP.TXT)
```

-- set the directories so you can find all the proper files...

```
;;;(SETQ DIRECTORIES '(
;;;"{DSK}/home2/will/sybalsky/lispcore/Sources/"
;;;          "{DSK}/home2/will/sybalsky/lispcore/library/"
;;;          "{DSK}/home2/will/sybalsky/lispcore/internal/library/"
;;;"{dsk}/home2/will/sybalsky/lispcore/sunloadup/"))

(DRIBBLE "{DSK}SUNLOADUP/LOADUP.LOG")

;(SETQ DIRECTORIES '(
;"{DSK}~/SUNLOADUP/lispcore/Sources/"
;           "{DSK}~/SUNLOADUP/lispcore/library/"
;           "{DSK}~/SUNLOADUP/lispcore/internal/library/"
"{dsk}~/SUNLOADUP/lispcore/sunloadup/"))

(SETQ DIRECTORIES '(
"{DSK}/users/sybalsky/lispcore/Sources/"
            "{DSK}/users/sybalsky/lispcore/library/"
            "{DSK}/users/sybalsky/lispcore/internal/library/"
"{dsk}/users/sybalsky/lispcore/sunloadup/"))
```

--you really want the source code for this

```
(LOAD 'FILESETS)
```

```
(FILESLOAD RENAMEFNS MAKEINIT DLFIXINIT CMLARRAY-SUPPORT)
(LOAD "{DSK}/users/sybalsky/FASTINIT.DFASL")

-- turn off idle or you get stuck.

(IDLE.SET.OPTION 'TIMEOUT T)

-- and start making the init.  This takes about 2.5 hrs.

;(PROGN
;  (DORENAME 'I)
;  (DLFIXINIT
;   (MAKEINIT '(11500Q 13062Q 25400Q)
;      NIL NIL
;      '({DSK}/home2/will/sybalsky/lispcore/Sources/
{dsk}/home2/will/sybalsky/lispcore/sunloadup/ ))
;   '{DSK}INIT.DLINIT
;   '{dsk}/medley/project4/venue/LISPDLION.DB
;   300)
;  (COPYFILE '{eris}<lispcore>sunloadup>XREM.CM '{DSK}XREM.CM)
;  (COPYFILE '{eris}<lispcore>sunloadup>LOADUP-REM.CM '{DSK}LOADUP-REM.CM)
;  (LOGOUT T)
;)
(PROGN
  (DORENAME 'I)
  (DLFIXINIT
   (MAKEINIT '(11500Q 13062Q 25400Q)
      NIL NIL
      '({DSK}/users/sybalsky/lispcore/Sources/
{dsk}/users/sybalsky/lispcore/sunloadup/ ))
   '{DSK}INIT.DLINIT
   '{dsk}/users/sybalsky/lispcore/next/LISPDLION.DB
   300)
  (COPYFILE '{dsk}/users/sybalsky/lispcore/sunloadup/XREM.CM
      '{DSK}SUNLOADUP/XREM.CM)
  (COPYFILE '{dsk}/users/sybalsky/lispcore/sunloadup/LOADUP-REM.CM
      '{DSK}SUNLOADUP/LOADUP-REM.CM)
  (DATE)
  (DRIBBLE)
  (LOGOUT T)
)

-- Now, if you are on the "loadup" machine, exit medley and go to another
-- machine.  RLOGIN to the loadup machine and do the following:

-- Build an init-specific lde note: you must have a directory under
-- the maiko directory called init.ARCH where ARCH is the architecture
-- of the machine you will run the lde on.  On a sun4, it would be init.sparc.

cd ~/maiko/bin
makeinitlde -e

-- connect back to your home directory and make a link to the lde and
-- ldeether (fill in yourname and machine os and arch in the proper
-- slots below

cd ~
ln -s /users/YOURNAME/maiko/init.ARCH/lde
ln -s /users/YOURNAME/maiko/init.ARCH/ldeether

-- make sure you don't have LDEDESTSYSOUT set as you want the sysout on your
home
-- directory.
```

```
-- You will need dbxinit.txt available

-- YOU MUST USE A FRESH COPY OF XREM.CM EVERY TIME YOU TRY THIS AS IT
-- GETS SMASHED AT STARTUP

cp ~/XREM.CM ~/REM.CM

-- start lde under dbx

-- init lde can't treat 'LDEDESTSYSOUT'
unsetenv LDEDESTSYSOUT

dbx lde

-- load the dbxinit

source /users/maiko/working/bin/dbxinit.txt

-- now set up to stop on error (before URAID, which loses 'cause
-- it can't find the keyboard.)

err

run ~/INIT.DLINIT -INIT -NF

-- this is going to run and eventually log itself out.  when dbx returns, quit
-- from dbx and presto! You've built the beginnings of a loadup.

---

-- go to the loadup machine and connect to the place where you normally
-- get your lde from

cd ~/maiko/sunos4.sparc/

-- Get the new REM.CM: (YOU MUST DO THIS EVERYTIME AS
-- REM.CM gets wasted on startup!!)

cp ~/LOADUP-REM.CM ~/REM.CM
ldeether

-- Now this is going to march happily through loading files.  When it turns
-- on the windowworld, you may have to hit the space bar to make it continue.

-- I don't know how to make PUP and LEAF load at this point, but I'm working
-- on it.

-- I am also working on integrating the changes to the emulator with the
latest
-- stuff.

-- closure caching is still off.

-- Questions or comments?
```

HOW TO MAKE SYSOUT ON SUN

Osamu Nakamura:KSPA:Fuji Xerox
February 20, 1990

SUN [redacted] SYSOUT [redacted].

[redacted] SYSOUT [redacted] Dorado [redacted].
Dorado [redacted] FX [redacted] SYSOUT [redacted],
[redacted], Venue [redacted] Dorado [redacted] SUN [redacted]
SYSOUT [redacted].

[redacted], Venue [redacted] John D. Sybalsky [redacted],
SUN [redacted] SYSOUT [redacted].

[redacted]

1.MAKEINIT/LOADUP [redacted].
[redacted] ~/SUNLOADUP [redacted].

- runloadup.
- FILESETS.
- INIT.MAKEINIT
- XREM.CM;1
- LOADUP-REM.CM;
- LOADUP.LISP;

2. Medley [redacted]
   - [redacted] ([redacted] `LDE`)
   - INIT [redacted] ([redacted] `INITLDE`)

MAKEINIT [redacted] INIT.DLINIT [redacted]

3. LISP.SYSOUT [redacted]
(Lispcore/sources [redacted].)

4. Medley [redacted] (makefile [redacted])
([redacted] `INITLDE` [redacted])

[redacted]

1. `INITLDE` [redacted]

`INITLDE` [redacted] makefile [redacted] makeinitlde [redacted],
`INITLDE` [redacted].
`INITLDE` [redacted], [redacted],
[redacted] $YOURWORKDIR/init.$ARCH/ [redacted].
[redacted].

prompt% mkdir $YOURWORKDIR/init.$ARCH
-- $ARCH [redacted] sun4 [redacted]sparc , sun3 [redacted] mc68020 [redacted].
prompt% cd $YOURWORKDIR/bin
prompt% makeinitlde -e
-- [redacted] "-DINIT" [redacted].

INITLDE '$YOURWORKDIR/init.$ARCH/ lde' [redacted].

[redacted]: [redacted]INITLDE[redacted]
cg3,cg6 [redacted].
cg3,cg6 [redacted]INITLDE[redacted],
makeinitlde [redacted]OPTFLAGS [redacted] -DDISPLAYBUFFER [redacted] $YOURWORKDIR/init.$ARCH/ [redacted],
makeinitlde -e [redacted].

2. MAKEINIT/LOADUP [redacted]

MAKEINIT/LOADUP[redacted]Venue [redacted].
[redacted]Venue [redacted].

FILE: runloadup

```
set LDE              = $HOME/maiko/sunos4.sparc/lde
set LDEPATH          = $HOME/maiko/sunos4.sparc
set INITLDE          = $HOME/maiko/init.sparc/lde
set INITLDEPATH      = $HOME/maiko/init.sparc
set FULL_SYSOUT      = /usr/local/sysouts/FULL.SYSOUT
set FIRST_REM_CM     = $HOME/SUNLOADUP/XREM.CM
set SECOND_REM_CM    = $HOME/SUNLOADUP/LOADUP-REM.CM
```

FILE: INIT.MAKEINIT

```
DIRECTORIES [redacted]
FILESETS [redacted]LOAD [redacted]
FASTINIT.DFASL [redacted]LOAD [redacted]
MAKEINIT[redacted]4[redacted]
DLFIXINIT[redacted]3[redacted]
```

FILE: LOADUP-REM.CM

```
LOADUP.LISP[redacted]LOAD [redacted]
```

MAKEINIT/LOADUP [redacted]

1. [redacted]

MAKEINIT/LOADUP [redacted].

- ~/I-NEW, ~/I-NEW.LCOM
- ~/INIT.SYSOUT
- ~/INIT.DLINIT
- ~/lisp.virtualmem

- ~/REM.CM
- ~/SUNLOADUP/LOADUP.LOG

2. [redacted]

[redacted] SUNLOADUP [redacted] runloadup [redacted].

prompt% cd ~/SUNLOADUP
prompt% runloadup

[redacted] SUNLOADUP [redacted] LISP.SYSOUT [redacted].

SYSOUT [redacted]

LOADUP.LISP [redacted]FILESETS [redacted]
SYSOUT [redacted].

Ethernet [redacted]SYSOUT

LISP.SYSOUT [redacted]Ethernet [redacted]SYSOUT [redacted].

[redacted]:

[1] FILESETS[redacted]
1LISPSET [redacted] LLETHER [redacted]

[2] LOADUP.LISP[redacted]
DPUPFTP LLNS TRSERVER SPP COURIER NSPRINT CLEARINGHOUSE NSFILING INTERPRESS [redacted]

[3] XREM.CM[redacted]
[redacted]2[redacted]
(MOVD (QUOTE \ETHEREVENTFN) (QUOTE \ETHEREVENTFN-))
(MOVD (QUOTE NILL) (QUOTE \ETHEREVENTFN))

[4] runloadup [redacted]

[redacted]: SYSOUT [redacted].

D-Machine [redacted]SYSOUT

LISP.SYSOUT [redacted]D-Machine [redacted](D-Machine [redacted])[redacted]SYSOUT [redacted].

[redacted]:

[1] LOADUP.LISP[redacted]
DPUPFT DISKDLION DOVEINPUTOUTPUT DOVEDISK DOVEDISPLAY DOVEMISC DOVEETHER DOVEFLOPPY DSKDISPLAY FLOPPY [redacted]

[2] runloadup [redacted]

[redacted]: SYSOUT [redacted].

[redacted]SYSOUT

LISP.SYSOUT [redacted]Interlisp BYTE Compiler, XCL compiler [redacted]SYSOUT [redacted].

[redacted]:

[1] LOADUP.LISP[redacted]
DLAP BYTECOMPILER COMPILE FASDUMP XCL-COMPILER DPUPFT DISKDLION DOVEINPUTOUTPUT DOVEDISK DOVEDISPLAY DOVEMISC DOVEETHER DOVEFLOPPY DSKDISPLAY FLOPPY [redacted]

[2] runloadup [redacted]

[redacted]: [redacted]
Undefined function SPECVARS [redacted] Break [redacted].
SPECVARS [redacted] [redacted]COMPILE [redacted].

BACKGROUND-MENU-BUTTONS

By: Doug Cutting (Cutting.PA@Xerox.COM)

This document last edited on January 28, 1988.

INTRODUCTION

BACKGROUND-MENU-BUTTONS makes it easy for the Rooms user to make buttons which do the same things as entries on the background menu.

EXPLANATION

When this module is loaded a button labelled "Make Background Button" is placed on the screen. When this button is pressed a menu which looks just like the background menu is raised. Selecting an entry from this menu will create a button which does the same thing as this entry.

en·vōs
Envos Corporation
1157 San Antonio Road
Mountain View, California 94043
(415) 966-6200
California: (800) 824-6449
Continental U.S.: (800) 228-5325

September, 1988

Dear ROOMS User:

Attached is the documentation for the Rooms User's modules.

Note that most of these modules include source code. Thus besides being useful in themselves, these modules provide good examples of programmatic use of ROOMS.

Rooms User modules are user-contributed software, and as such are not supported Envos products. We merely distribute these modules in hope that they will prove useful.

Please direct questions, suggestions and problems with Rooms User's modules directly to their authors. We have included the ARPAnet addresses of the authors in the documentation for this purpose.

We encourage you to contribute more Rooms User's modules. The same policies which apply to Lisp User's modules apply here. For more information see the Lisp User's guidelines distributed with the Medley release.

Customer Support

>>Module Name<<

By: >>Your Name<< (>>Your net address<<)

Uses: >>Other modules necessary to run this one<<

>>Type INTERNAL here if the file is for Internal Use Only<<

This document last edited on >>DATE<<.

INTRODUCTION

>>This paragraph should be replaced by an overview of your module.<<

MODULE EXPLANATIONS

>>Functions, Variables, and Lisp Code Examples<<

It is usual to first give the name of a function, then describe its purpose and each of its arguments. When the name of a function is first given, it is set off like this:

(IMAGEFNSCREATE *DISPLAYFN IMAGEBOXFN PUTFN GETFN COPYFN*) [Function]

The function name is in 10-point regular Modern, all caps. Arguments are in 10-point italic Modern, in all caps, mixed case, or lowercase, as they appear in the system. Variables look like functions, except that the word ‘‘Variable,’’ enclosed in **square brackets**, follows the variable name. Please note that these are the characters [], not the parentheses

```
This is an example of code. It is in 10-point Terminal font.
```

Function names, commands, file names, and the like are in 10-point modern.

Be sure to include the following information in any module explanations:

• any file dependencies

• definitions of all arguments

• module, function variable, etc. limitations

• a liberal number of examples for all functions, variables, etc.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

HYPERDESK and HYPERDESK-INIT

The HyperDesk (or "MobyDesk") sysout is a "Swiss Army Sysout" that includes much of the best local functionality. This includes Sketch, DictTool, PCL, CommonLens, Rooms, and Notecards. The HyperDesk-INIT file provides a powerful init file for HyperDesk Users as well as those that may use other sysouts but would like convenient access to HyperDesk functionality. This Init file can make life easier for those who don't hack their own init files by making it easier on those that are called in to do so. A reasonable goal is that users of this class wouldn't need separate init files and instead will be able to tailor thier environment through a control panel and automatic profile saving.

Author: Ramana Rao
Filed on: {EG:PARC:XEROX}<RAO>HYPERDESK>HYPERDESK.TEDIT;5
Last filed: April 26, 1988 14:14 PDT (Tuesday)

DRAFT

HYPERDESK

The HyperDesk.SYSOUT is available from <LISP>LYRIC>BASICS> on your favorite Lisp Servers (i.e. IE: and PHYLUM). It contains the SKETCH, DICTTOOL, ROOMS, PCL, COMMON-LENS, and NOTECARDS.

\CC-HYPERDESK-MSG [Variable]

When HYPERDESK is greeted, it automatically sends a LAFITE message to the maintainer. By default this message is CC'ed to the person using the sysout; this CC'ing can be disabled by setting the variable \CC-HYPERDESK-MSG to NIL in the users Init File.

Loading HYPERDESK-INIT

The HYPERDESK-INIT file is available to PARC users of the Lyric release of Xerox Lisp. It can be found on the directory {EG:PARC:}<RAO>HYPERDESK>. Including the file command

```
(FILES (FROM LISPUSERS)
       "HYPERDESK-INIT")
```

in your personal INIT file will automatically load HYPERDESK-INIT when you start up a new Lisp.

HYPERDESK-INIT is essentially wrapping around GENERIC-INIT, so users of HYPERDESK-INIT should understand the essentials of GENERIC-INIT. Documentation for it is available from {EG:}<LANNING>LISP>USERS>GENERIC-INIT.TEDIT. Both of these centralized init files set variables and invoke operations that tailor the environment. Much of this behavior can be controlled or preempted by setting user parameters or special control variables either before or after the load of HYPERDESK-INIT. Two variables, LOGINHOST/DIR and HOME-MACHINE-NAME, must be set before the load of HYPERDESK-INIT (or at least setting them should be considered explicitly). These are used as control variables to GENERIC-INIT and are documented there.

An example of a minimal init file can be found on {EG:}<RAO>HYPERDESK>LYRIC-INIT. Any of these INIT files (HYPERDESK-INIT, GENERIC-INIT, LYRIC-INIT, my {EG:}<RAO>LISP>INIT-LYRIC or Stan's {EG:}<LANNING>LISP>LYRIC-INIT) can be loaded prop into your sysout without any thing happening so you can see what would happen. The one caveat is that they will smash filecoms of files with the same name.

Tailoring HYPERDESK-INIT

There are a number of variables that can be set before and after loading HYPERDESK-INIT to tailor its behavior to ones liking. In addition, users can also tailor GENERIC-INIT by setting variables documented in the GENERIC-INIT documentation. A few of the GENERIC-INIT variables (*LOAD-UTILITY-OPTIONS*) are used by HYPERDESK-INIT, so users should use the variables provided by HYPERDESK-INIT as described below before resorting to GENERIC-INIT variables. Unless otherwise noted, all symbols described are in the INTERLISP package.

Things you MAY do before loading

HYPERDESK-FULL-INIT-P [Variable]

When HYPERDESK-INIT is loaded, it will ask if it should do a full initialization. By default this message will time out, set this variable and continue to a full initialization. This is especially convenient for starting non-hyperdesk sysouts where you don't want to wait for special status hyperdesk functionality to be loaded.

HYPERDESK-BASE-MODULES [Variable]

HYPERDESK-MODULES [Variable]

HYPERDESK-EXTRA-MODULES [Variable]

These variables control what modules are loaded automatically by GENERIC-INIT. HYPERDESK-INIT initvars *LOAD-UTILITY-OPTIONS* to the append of these three variables in the order listed. *HYPERDESK-MODULES* is initilized to the special status hyperdesk modules (ROOMS, PCL, and COMMON-LENS, and a bunch of window types for ROOMS) unless *HYPERDESK-FULL-INIT-P* is NIL in which case *LOAD-UTILITY-OPTIONS* will not be touched. Note that these should all be lists of strings where the strings are the name of the package as seen on the Load Utility Menu (i.e. the name of the top level or only file of the module e.g. LoadPCL or COMMON-LENS). See GENERIC-INIT *LOAD-UTILITY-OPTIONS*.

HYPERDESK-SUITES-P [Variable]

HYPERDESK-AX-POCKETS-P [Variable]

HYPERDESK-AX-ORIGINAL-P [Variable]

HYPERFLOOR-SUITE-P [Variable]

HYPERDESK-SUITE-P [Variable]

HYPERDESK-USER-SUITES [Variable]

HYPERDESK-INIT provides some mechanisms for setting up initial suites. Setting *HYPERDESK-SUITES-P* to NIL will disable this facility altogether. Setting *HYPERDESK-AX-POCKETS-P* and *HYPERDESK-AX-ORIGINAL-P* to NIL will disable deletion of POCKETS and ORIGINAL rooms respectively. Setting *HYPERFLOOR-SUITE-P* and *HYPERDESK-SUITE-P* to NIL will disable loading of these two default suites respectively (note that disabling HYPERFLOOR suite will also disable the HYPERDESK Suite). Finally *HYPERDESK-USER-SUITES* should be a list of fully qualified file names that determine which suites get loaded initially.

HyperFloor and HyperDesk Suites

HyperFloor and HyperDesk Suites provide a basic framework for working with Rooms and other HyperDesk functionality.

HyperFloor provides three rooms: HyperPanel which is basic inclusion room, Help where eventually various buttons for getting documentation will be put, and the Boiler Room where buttons for fixing things and dealing with the HyperDesk in general will be put. One particularly important button in the Boiler Room is one that allows you to reload the HyperFloor so you can import any recent additions to the Boiler or Help Rooms.

HyperDesk provides two rooms: Mail Room and Office. Users should copy these rooms to make suites of their own and avoid working in these if they want to build their own suite.

LAFITE-WINDOW-TYPES

By: Doug Cutting (Cutting.PA@Xerox.COM)

This document last edited on January 28, 1988.

INTRODUCTION

This module provides window types for Lafite windows. Lafite is the Interlisp-D mail program. Loading this module enables you to save Lafite windows in your suite files.

EXPLANATIONS

We provide window types for Lafite browsers and for the Lafite status window.

Note: we do not provide window types for message windows, as these are far more transient than the status window and browsers.

OFFICE.SUITE

By: Doug Cutting (Cutting.PA@Xerox.COM)

This document last edited on January 28, 1988.

INTRODUCTION

This is a sample suite file.

Beginning Rooms users might load this, augment it as desired, and then use the "Dump Suite" command to save their own version of it.

This suite also gives some ideas about how one might use the facilities which Rooms provides.

INSTALLATION

Load the file OFFICE.SUITE. This file (like all suite files) will load ROOMS if it is not already loaded. It will then create the rooms in the OFFICE suite. Use the Overview to see these rooms.

RANDOM-WINDOW-TYPES

By: Doug Cutting (Cutting.PA@Xerox.COM),
Stan Lanning (Lanning.PA@Xerox.COM) and
Ramana Rao (Rao.PA@Xerox.COM)

This document last edited on April 21, 1988.

This module contains window type definitions for many LispUsers modules.

These definitions serve two purposes. Their primary purpose is to allow one to save windows created by these modules in suites. Secondarily they provide good examples of window type definitions for the programmer attempting to define his own.

Modules covered are:

WHO-LINE
CALENDAR
PRINTERMENU
CROCK
BICLOCK
ADDRESSBOOK
PHONE-DIRECTORY
GRID-ICONS

In addition, support is provided for the FILEWATCH LispUsers module. This works by adding a property to each room which notes whether FILEWATCH is on in that room. This property is not inherited, i.e. including a room in which FILEWATCH is does not turn FILEWATCH on.

A listing of RANDOM-WINDOW-TYPES can be found in Appendix A of this document.

>>Module Name<<

By: >>Your Name<< (>>Your net address<<)

Uses: >>Other modules necessary to run this one<<

This document last edited on >>DATE<<.

INTRODUCTION

>>This paragraph should be replaced by an overview of your module.<<

MODULE EXPLANATIONS

>>Functions, Variables, and Lisp Code Examples<<

It is usual to first give the name of a function, then describe its purpose and each of its arguments. When the name of a function is first given, it is set off like this:

(IMAGEFNSCREATE *DISPLAYFN IMAGEBOXFN PUTFN GETFN COPYFN*) [Function]

The function name is in 10-point regular Modern, all caps. Arguments are in 10-point italic Modern, in all caps, mixed case, or lowercase, as they appear in the system. Variables look like functions, except that the word ‘‘Variable,’’ enclosed in **square brackets**, follows the variable name. Please note that these are the characters [], not the parentheses

```
This is an example of code. It is in 10-point Terminal font.
```

Function names, commands, file names, and the like are in 10-point modern.

Be sure to include the following information in any module explanations:

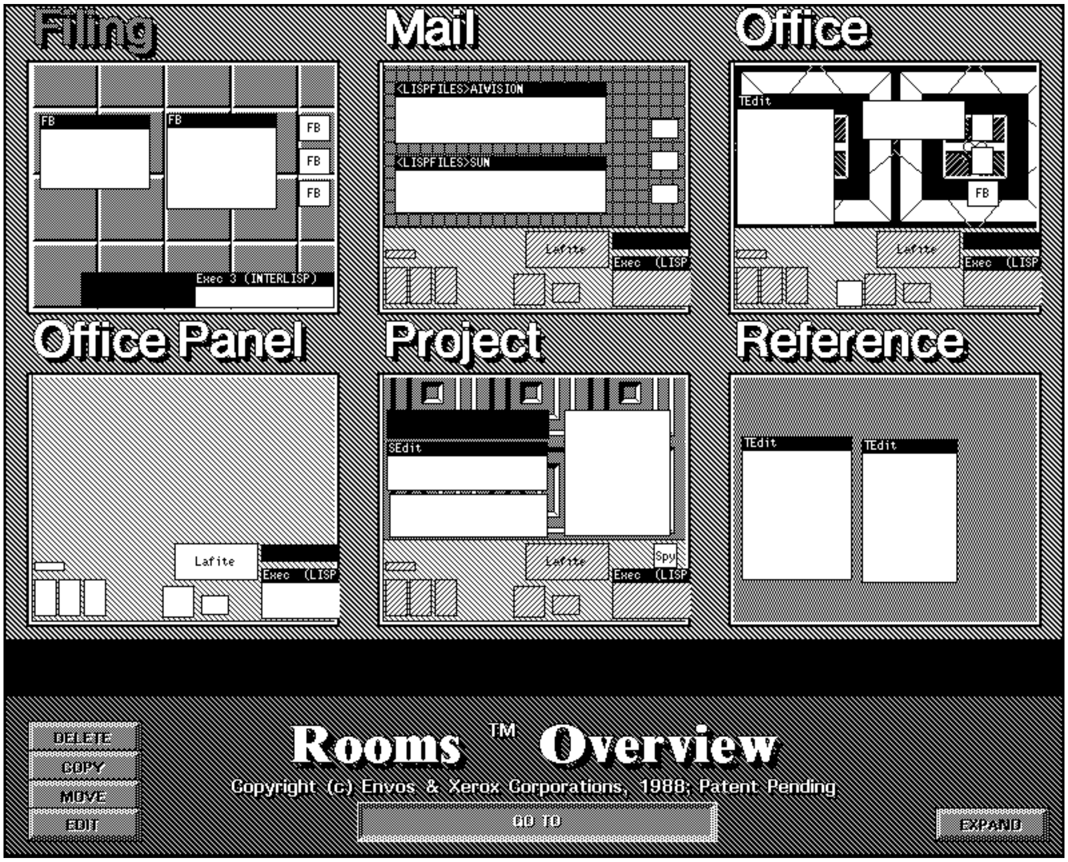
• any file dependencies

• definitions of all arguments

• module, function variable, etc. limitations

• a liberal number of examples for all functions, variables, etc.

ROOMS™



Overview

ROOMS is a powerful interface to the Medley lisp environment window management system. ROOMS effectively increases the size of the screen and allows you to organize the work environment to facilitate management of complex parallel tasks.

For example a user may be developing and debugging a program, writing a paper, doing background research, reading mail, and filing all more or less at the same time. With limited screen space a user performing all these activities looses productivity shuffling through windows, icons, and other assorted screen clutter in order to reestablish the context of a task. ROOMS solves this problem of context switching by allowing users to create workspaces (called rooms), each one analagous to another screen, and containing only those tools needed for a specific task. Moreover, ROOMS provides methods for easily moving from one room to another and customizing the "look" of each room further aiding in efficient task-switching.

By providing dedicated workspaces, ease of navigation between workspaces, and a graphical link between each workspace and its related task, ROOMS helps users minimize context recovery time caused by task switching. In addition ROOMS provides an easy means to develop highly graphical custom interfaces for end-user applications. The overall result of this seamless integration of form and functionality is increased productivity in performing tasks in the Envos Software Development Environment.

Research Background

The ROOMS system design resulted from the following observations by members of the Xerox Palo Alto Research Labs:

1) Most intellectual work requires coordinating many sources of information, e.g. notes, drafts, spreadsheets, program listings, etc.

2) Before the widespread dissemination of desktop workstations and personal computers, knowledge workers typically spread paper containing these items out on a desk (or dining room table). But the small screens of computer workstations do not allow such a lavish use of space when working on many concurrent tasks. For example, it takes 22 average PC screens or 10 19" workstation screens to equal the area of a typical desk.

3) The result for overlapped windows systems is a kind of electronic messy desk, where the user spends large amounts of time moving, shrinking, and resizing windows in order to switch tasks.

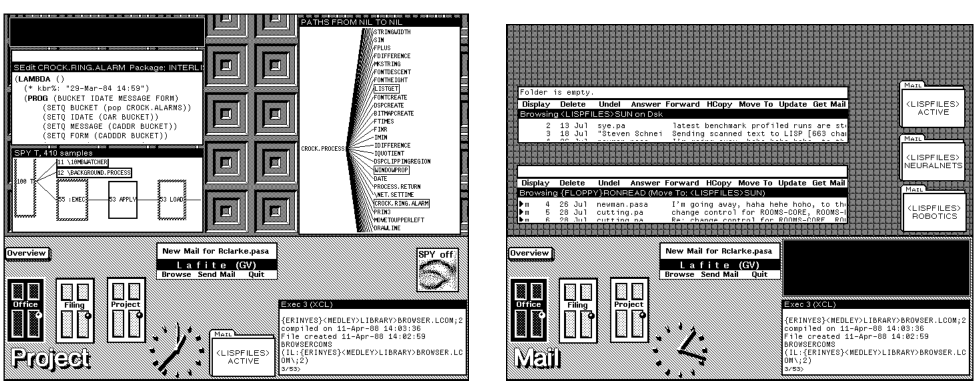
4) An interface designed to assist a user to work on concurrent tasks rather than assume linear work habits would have the following features:

   - Fast task switching and fast task resumption
   - Easy to re-acquire mental task context
   - Access to a large amount of information
   - Fast Access to information
   - Low user overhead
   - Engaged tools shareable among several tasks
   - Collections of engaged-tools shareable among tasks
   - Task-specific presentation of shared-engaged tools.

How Envos' implementation of ROOMS accomplishes these goals is described in the next section "The ROOMS Design".

The ROOMS Design

Rooms solves the problem of user task switching by allowing users to create a number of screen-sized workspaces called Rooms. The figure below shows two Rooms.



Two example Rooms (a) used for programming and (b) used for reading mail

In each room, there are a number of small icon-like objects called Doors. When a door is selected with the mouse, the user has the illusion of transiting to a new room containing windows. Each room is related to a different major task, such as working on a particular project, reading mail or file management. In the room are a number of engaged tools related to the task. When a user re-enters a room it appears exactly as when it was left.

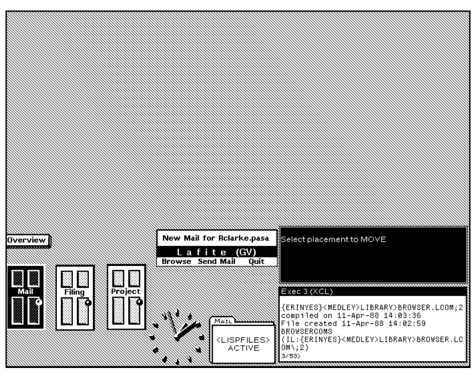
The basic notion of the Rooms scheme is simple. But in order for this basic notion to be successful the ROOMS system design has many features to help manage window interaction among rooms and to keep the user from getting lost.

Task Interaction

When working in ROOMS, there are usually a number of engaged tools that the user wants to share among different tasks. Examples of these might include a code editor, a text editor or a clock. The following ROOMS design features allow for the sharing of engaged tools:

Placements A window in a room has a specific location and shape. This is known as a Placement. The same window can exist in several rooms, with different locations, shapes, etc., in each room. Actions done on a shared window in one room are reflected in another room. Note that the black promptwindow in the Mail and Project rooms has a different location and shape in each room.

Inclusions Rooms can be included within other rooms. This allows collections of rooms to share a common set of windows, known as Inclusions. The band of windows and icons common in both parts of the Mail and Project rooms is a control panel, shown below, implemented as an included room.



Example of an included room called Office Panel

Baggage When moving between rooms you may carry windows with you. For example, you may wish to bring program code from one workspace to a workspace where you are doing documentation. This is accomplished by holding down the Move or Copy key when selecting a door. You will then be prompted to select the windows you wish to carry with you to the next room. The windows will have the same presentation in the new room.

Pockets You can also have a constant piece of baggage called a Pocket. A Pocket is a room dynamically included in all rooms. Whichever windows are placed in your Pocket (a clock say) will automatically occur (at the same location and presentation attributes) in all rooms.

Navigation

The organization of the user's workspace into a number of workspaces creates a potential navigational problem. ROOMS solves this problem with a number of navigational aids.

Doors Doors provide the basic link mechanism between rooms. Selecting a door with the mouse "moves" you to the new room containing windows associated with that room.

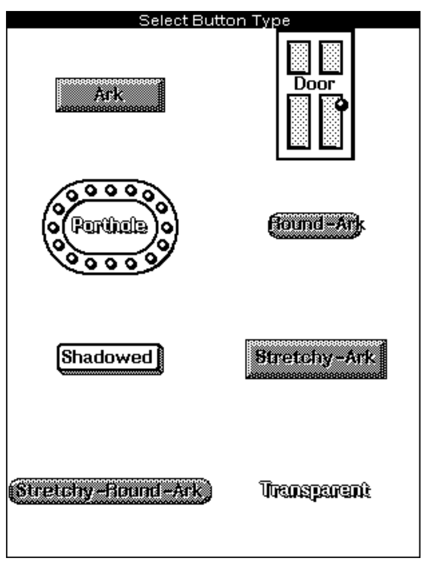
Back Doors Rooms allows you to create a Back Door in a room. If you have created a Back Door you can easily move back to the task you were previously working on before you entered the present room.

Rooms Menu Selecting the Go To Room sub-item from the ROOMS background menu presents you with a list of Room names. Selecting one will move you to that Room.

Overview The main feature of the Overview is a set of Room pictographs as shown in the openning diagram. From the Overview you can see the overall layout of a room and the tools it contains. The room pictographs can be instantly expanded one at a time, allowing you to browse through the windows in the entire set of rooms. In addition you can enter any room directly via the Overview.

Tailorability

Several mechanisms are provided to help users tailor rooms. First, simply creating, moving, deleting and shaping windows in the usual way causes things to exist in rooms. Thus ROOMS preserves the natural interaction with the user. Second, special background entries are provided to allow the user to create new doors and other conveniences of construction. Below is an example of the standard set of Doors available to users.



At the Overview level, it is possible to copy, move or delete window pictographs within a room and between rooms and have the changes reflected in the rooms themselves. And finally, Rooms has a simple layout language for creating unique backgrounds for Rooms. By using the structure editor, SEdit, on this layout language, users can run arbitrary procedures on the entrance and exit of a room and can compute specialized backgrounds for rooms. Below is an example of a SEdit of a rooms layout expression for the Project room shown earlier.

```
SEdit Project  Package: XCL-USER
(:INCLUSIONS ("Office Panel") :BACKGROUND
   ((:REGION (0 1/4 1.0 3/4) :SHADE
       (:EVAL ROOMS:RENAISSANCE-BITMAP)
       :BORDER 2)
    (:TEXT "Project" :FONT
           (IL:HELVETICA 36 IL:BOLD)
           :POSITION (10 . 10))))
```

Suites Having designed an environment to suit your needs, ROOMS allows you to restore that state if there is a need to reload your system. Subsets of rooms can be grouped together to form Suites. Suites can be saved to a file and later restored to a new enviroment or shared with other users. Most system windows have a ROOMS window-type specification which allows them to be saved and restored from a suite. Thus a Filebrowser window that you have open on a particular system can be saved and the window placement and contents will be restored when you reload the suite onto a new system.

Buttons Buttons are a unique user-interface device that provides for the execution of any lisp command at the click of the mouse. Buttons can be of any shape and size. Doors are examples of buttons that move you from one room to another. For example selecting the following button will perform a directory listing of the local disk.

The button definition is shown below:

```
SEdit Directory  Package: XCL-USER
(:TEXT "Directory" :FONT
       (IL:HELVETICA 12
                     (IL:BOLD IL:REGULAR
                              IL:REGULAR))
       :SHADOWS :ARK :TYPE :STRETCHY-ROUND-ARK
       :HELP "Bring up local disk directory"
       :ACTION (DIR "{dsk}<lispfiles>")
       :INVERTED? NIL)
```

Customization of Rooms

When used as the interface to an application, ROOMS provides you with a complete programmatic interface to allow application specific customization. This includes the ability to have applications create and switch rooms under programmatic control as well as enable users to design custom buttons and window-types.

References

A Multiple, Virtual-Workspace Interface to Support User Task Switching, Austin Henderson/Stuart Card, 1987 ACM CHI + GI Conference Proceedings on Human Factors in Computing Systems and Graphic Interfaces

For further information about ROOMS contact your Envos Marketing Representative, or call toll-free in the Continental United States 1-800-228-5325, or in California 1-800-824-6449.

Envos Corporation

1157 San Antonio Road

Mountain View, California 94043

(415) 966-6200

ROOMS USERS' RULES

This document describes the rules and procedures for "Rooms Users" modules. This document is mainly for Rooms Users' module writers, but users should also understand the rules.

DEVELOPING A ROOMS USERS' MODULE

A Rooms Users' module is a useful program made available to the general Rooms community. Neither the author nor the custodian of Rooms Users' imputes any warranty of suitability or responsibility for errors.

Rooms Users' modules should be easily distinguishable from released library and Lisp Users' modules. In particular, this means that a Rooms Users' module may not have the same name as a Library or Lisp Users' module and should be visibly different. Rooms Users' modules derived from released software should be announced to the public only after communicating with the organization responsible for that released software.

Testing is important. If you make significant changes to a Rooms Users' module, enlist users at your site as alpha testers. A Rooms Users' module is not shoddy software; it is software made available outside the regular release channels.

ROOMS USERS' MODULE OWNERSHIP

A module submitted for Rooms Users' remains the "property" of the submitter. Others may not make changes, except for their own private use, without negotiating with the owner (who may already be making similar or incompatible changes).

As the owner of a module, you are not required to fix bugs, but if not, you must be willing to transfer ownership (permanently) to someone who volunteers to fix them. Ownership may pass back and forth among several people as long as they agree.

SUBMITTING ROOMS USERS' MODULES

If you are not an internal user, you should submit your new module to us through e-mail or on a floppy or tape. External users should make sure that they include all relevant information, such as documentation containing an e-mail or US mail address where he/she can be reached.

SUBMITTING FILES TO ROOMS USERS'

As with released software, it is important to submit not just the resulting product, but all the files needed to build and maintain a Rooms Users' module:

1. the file to load (.LCOM or .dfasl or .SUITE)
2. documentation describing it, following the formatting rules (see below)
3. a source file that can be released (optional)
4. data files, if needed

Modules submitted once are released once. Do not assume that a module submitted for one release will be automatically released in subsequent releases.

DOCUMENTATION

No modules will be released without documentation. Documentation can be as simple as a paragraph describing what the module does and how to use it, or it can be as extensive as a dozen-page user manual. All modules should have a file with a .TEDIT extension. Formatting should be done according to the rules outlined in the Rooms Users' Template, included on the Rooms Users floppy or tape as EASYTEMPLATE.TEDIT and also printed in this document. All users, external users included, should follow the Rooms Users' Template rules. If the documentation is large and formatting time consuming, you can also produce an interpress file (with the .ip extension), as well as submitting a .TEdit file. (Be sure to update the interpress file if you update the documentation!) Documentation should include the full address of the submitter.

COMPATIBILITY

Any submitted Rooms Users' files should be compilable in a "vanilla" Rooms environment. The file itself should load in any auxiliary modules when necessary.

Thanks for your cooperation.

Table of Contents

[This page intentionally left blank]

ROOMS USER'S MODULES

en·vōs

300002
Medley Release
SEPTEMBER 1988

Address comments to:
ENVOS
User Documentation
1157 San Antonio Rd.
Mountain View, CA 94043
415-966-6200

ROOMS USER'S MODULES

PART NUMBER 300002

SEPTEMBER 1988

Text was written and produced with ENVOS text formatting tools; Xerox printers were used to produce text masters. The typeface is Optima.

TOUCHY BUTTONS

By: Ramana Rao (Rao.pa)

Uses: Rooms

This document last edited on January 22, 88.

INTRODUCTION

This Rooms Users Package provides a number of "touchy" buttons i.e. buttons that visually depend on the state of the world and change the state of the world when touched. Right now I provide three types of touchy buttons: includer, toggler, and once-only. You can make a touchy button by calling rooms::make-<touchy-type>. I will take suggestions for any others that people think may be useful since I'm trying to abstract the touchy technology.

BUTTON CONSTRUCTORS

(ROOMS::MAKE-INCLUDER *ROOM-NAME*) [Function]

Includers allow you to conveniently mixin or mixout rooms. For example, you can have "Notecards-Mixin" and "Programming-Mixin" Rooms and have includer buttons in your personal "Pockets." Then you can include these functionality traits whenever you need them no matter the room.

(ROOMS::MAKE-TOGGLER *VARIABLE-NAME*) [Function]

Toggle buttons allow you to toggle boolean variables. This should obviously be generalized to something that allows you to select or circulate through value settings.

(ROOMS::MAKE-ONCE-ONLY *FORM INITIAL-TEXT FINAL-TEXT*) [Function]

A once-only button evaluates a form exactly once in a sysout and then displays that it is exhausted. For Example:

```
(ROOMS::MAKE-ONCE-ONLY (IL:PROMPTPRINT "Hello, World") "Fire..."
"Exhausted")
```

UN-HIDE-TTY

By: Doug Cutting (Cutting.PA@Xerox.COM)

This document last edited on April 28, 1988.

ROOMS changes the window system such that open windows are not always visible. Some applications presume that open windows are visible. This module attempts to amend this situation.

Often when an application decides to start using a window which is not visible it gives that window the caret. CHAT is an example of such an application. When IL:CLOSECHATWINDOWFLG is NIL (the default) CHAT windows are left open after their connection is closed. If one leaves such a CHAT window in another room and then attempts to open a CHAT connection, CHAT will merrily re-use the hidden window. But as CHAT takes the caret when it opens a connection, we can identify that window and pull it into the current room.

(ROOMS:UN-HIDE-TTY) [Function]

If the window with the caret is not in the current room, it is brought into the current room with a call to ROOMS:UN-HIDE-WINDOW. If the window with the TTY is already visible on the screen then it is flashed. If there is no TTY window (i.e. the process with the keyboard has no TTY window) then a message to this effect is printed to the prompt window.

One can use this function in the action of a button, e.g.:

```
(ROOMS:MAKE-BUTTON-WINDOW
  (ROOMS:MAKE-BUTTON :TEXT "Un-Hide TTY" :ACTION '(ROOMS:UN-HIDE-TTY)))
```

Control-Y [Interrupt]

Brings the window with the caret into the current room by calling ROOMS:UN-HIDE-TTY. This interrupt is installed when UN-HIDE-TTY is loaded. This will not work with applications which have their own interrupt tables, e.g. TEdit and CHAT.

ROOMS USERS' MODULES

INTRODUCTION

ROOMS User's modules are written by users of Envos ROOMS. They are supported by the individuals who wrote them, not by Envos. Envos takes no responsibility for the reliability and maintenance of these modules.

Each module is documented, some in detail, others with only enough information to get the user started. At the top of each document, the name of the author is written, along with his electronic mail or U.S. mail address. Please contact the author with any problems you may have.

Note that most of these modules include source code. Thus besides being useful in themselves, these modules provide good examples of programmatic use of ROOMS.

This document contains explanations of each module, written by the author of each. It also contains an explanation of how to submit your own ROOMS User' module. Please read these instructions carefully if you wish submit a module.

WALLPAPER

By: Doug Cutting (Cutting.PA@Xerox.COM) and
Larry Masinter (Masinter.PA@Xerox.COM)

Uses: SCREENPAPER

This document last edited on August 8, 1988.

INTRODUCTION

This module provides an easy way to create distinctive backgrounds for your rooms.

All symbols described in this document are in the package ROOMS.

FUNCTIONS

(MAKE-WALLPAPER-WINDOW &OPTIONAL REGION) [Function]

Makes and returns a Wallpaper window. When the LEFT or MIDDLE mouse button is pressed over Wallpaper windows the user is asked to select a tile size, then a position for the tile. Positions are selected with the LEFT button. Each tile is displayed in the window. If the user presses the MIDDLE button then the background of the current room is changed to be the current tile. The user can abort this process at any time by pressing the RIGHT mouse button.

There is a window type definition for Wallpaper windows so they may be saved in Suites.

(HACK-BACKGROUND SHADE &OPTIONAL ROOM) [Function]

Changes the first background shade specified for ROOM to be SHADE. If ROOM does not paint the background then this function adds a command to the background specification for ROOM which paints the whole screen with SHADE. ROOM defaults to the current room.

New architecture for character input-output and alternative external formats

Ron Kaplan, May 2021

The Medley system was built with the Xerox Character Coding standard as the target for multi-byte input and output and for the internal mapping of character codes to glyphs.

This is now quite out of date, and our goal is to move to more modern conventions like Unicode and UTF-8.

The coding conventions are embodied in macros that test a stream to see if it is XCCS, and to do special open-coded processing (often with the help of locally bound variables for encoding information) if it is.

If it isn't XCCS, then the macros instead apply functions that are obtained from fields in the stream. This is optimized for the default XCCS set up because in that case a separate function call is avoided, the action itself is open coded.

The new architecture recognizes that there may be an advantage to specifying a system default for character processing that avoids function calls but that doesn't depend on support (binding of special variables as opposed to accessing stream fields on each call) to get that last measure of efficiency.

Thus, there are 4 generic macros corresponding to the 4 character IO operations:

\INCCODE
\OUTCHAR
\BACKCHAR
\PEEKCCODE

Each of these is defined to fetch a corresponding field from the stream (OUTCHARFN, INCCODEFN, PEEKCCODEFN, BACKCHARFN). If that field is NIL, then each of these passes to a corresonding default macro:

\DEFAULTINCCODE
\DEFAULTOUTCHAR
\DEFAULTBACKCHAR
\DEFAULTPEEKCCODE

These default macros can then be redefined to make a wholesale switch of the default encoding standard.

The macro \OUTCHAR, for example, is defined as
    if the stream has an OUTCHARFN, apply it. Otherwise do the
\DEFAULTOUTCHAR
and so on for each of the others.

For the current XCCS default, \DEFAULTOUTCHAR is defined to call \XCCSOUTCHARFN.

The corresponding stream fields can be set directly, but the preferred interface is to wrap up the 4 functions for a given format in an EXTERNALFORMAT datastructure. The function

    (\EXTERNALFORMAT stream formatname)

applies the information in the format into the stream. A particular (non-default) format can be specified as an optional parameter when a stream is opened, and each file device can have its own default external format. Then there is also a variable that holds the name of the name of the system-wide default, currently :XCCS.

If the default external format is applied to a stream, the relevant function fields are set to NIL to kick off the default macro for that particular function, otherwise the function is copied from the external format to the stream.

An external format has the following fields:

NAME
INCCODEFN
PEEKCCODEFN
BACKCHARFN
OUTCHARFN
EOL

The function (\INSTALL.EXTERNALFORMAT format) registers the given format under its name, so it can be retrieved when the name is given to \EXTERNALFORMAT.

If EOL is not NIL, then it is an end-of-line convention that will override whatever a stream might have had by default. (The value of EOL is one of the constants LF.EOLC, CR.EOLC, CRLF.EOLC.)

The system now includes external formats for
:XCCS (the global default)
:THROOUGH (untransformed bytes)

It probably would make sense to also include a :KEYBOARD external format, to generalize that as well.

UNICODE defines external formats for UTF8 with or without character translation, and also UTF16 (big-end and little-end). When we finally make the swap, we would make :UTF8 be the default, redefine the macros, and recompile all the callers.

The Japanse external formats that used to be included in the basic system are now provided by a JAPANESE in the library.

Finally, there is another macro \INCHAR that applies \CHECKEOLC to the result of \INCCODE.

FREE MENU

Free Menus are powerful and flexible menus that are useful for applications that need menus with different types of items, including command items, state items, and editable items. A Free Menu lives in a window, which can be opened and closed as desired, or attached as a control menu to the application window.

Making a Free Menu

A Free Menu is built from a description of the contents and layout of the menu. As a Free Menu is simply a group of items, a Free Menu Description is simply a specification of a group of items. Each group has properties associated with it, as does each Free Menu Item. These properties specify the format of the items in the group, and the behavior of each item. The function FREEMENU takes a Free Menu Description, and returns a closed window with the Free Menu in it.

Probably the easiest way to make a Free Menu is to define your own function which calls FREEMENU with the Free Menu Description right there in your function. This function can then also set up the Free Menu window as required by the application. The Free Menu Description is then saved as part of your function when you save your application.

Alternatively, you can save the Free Menu Description as a variable in your file, and then just call FREEMENU with the name of the variable. This may be a more difficult alternative if you want to use the backquote facility to built your Free Menu Description. See the section **Free Menu Item Descriptions**.

Free Menu Formatting

A Free Menu can be formatted in one of four ways. The items in any group can be automatically layed out in rows, in columns, or in a table, or else the application can specify the exact location of each item in the group. Additionally, Free Menu keeps track of the region that a group of items occupies, and items can be justified within that region. This way an item can be automatically positioned at one of the nine justification locations, top-left, top-center, top-right, middle-left, etc.

Free Menu Description

A Free Menu Description, specifying a group of items, is a list structure. The first thing in the list is an optional list of the properties of this group of items, in the form:

```
(PROPS <PROP> <VALUE> <PROP> <VALUE> ...)
```

The key word PROPS determines whether or not the optional group props list is specified. The section **Free Menu Group Properties** describes each group property. For now, the important property is FORMAT. The type of formatting determines the syntax of the rest of the Free Menu Description, in a very simple way.

When using EXPLICIT formatting, the rest of the description is any number of Item Descriptions, which have LEFT and BOTTOM properties specifying the position of the item in the menu. The syntax is:

```
((PROPS FORMAT EXPLICIT ...)
 <ITEM DESCRIPTION>
 <ITEM DESCRIPTION> ...)
```

When using ROW or TABLE formatting, the rest of the description is any number of item groups, each group corresponding to a row in the menu. These groups are identical in *syntax* to an EXPLICIT group description, with an optional PROPS list and then any number of Item Descriptions, except that the items need not have LEFT and BOTTOM properties, as the location of each item is figured out by the formatter. But the order of the rows and items is important. The menu is layed out top to bottom by row, and left to right within each row. The syntax is (the comments are not part of the description):

```
((PROPS FORMAT ROW ...)          ; props of this group
 (<ITEM DESCRIPTION>             ; items in first row
  <ITEM DESCRIPTION> ...)
 ((PROPS ...)                    ; props of second row
  <ITEM DESCRIPTION>             ; items in second row
  <ITEM DESCRIPTION> ...))
```

When using COLUMN formatting, the syntax is identical to that of ROW formatting. However each group of items corresponds to a column in the menu, rather than a row. The menu is layed out left to right by column, top to bottom within each column.

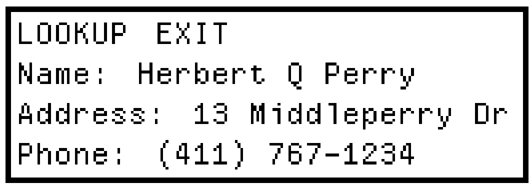
Finally, a Free Menu Description can have recursively nested groups. Anywhere the description can take an Item Description, it can take a group, marked by the key word GROUP. A nested group inherits all of the properties of its mother group, by default. However, any of these properties can be overridden in the nested groups PROPS list, including the FORMAT. The syntax is:

```
(                                ; no PROPS list, default row format
 (<ITEM DESCRIPTION>             ; first in row
  (GROUP                         ; nested group, second in row
     (PROPS FORMAT COLUMN...)    ; optional props
     (<ITEM DESCRIPTION> ...)    ; first column
     (<ITEM DESCRIPTION> ...))
  <ITEM DESCRIPTION>))           ; third in row
```

Here is an example of a simple Free Menu Description, for a menu which might provide access to a simple data base:

```
(((LABEL LOOKUP SELECTEDFN MYLOOKUPFN) (LABEL EXIT SELECTEDFN MYEXITFN))
 ((LABEL Name: TYPE DISPLAY) (LABEL "" TYPE EDIT ID NAME))
 ((LABEL Address: TYPE DISPLAY) (LABEL "" TYPE EDIT ID ADDRESS))
 ((LABEL Phone: TYPE DISPLAY)
  (LABEL "" TYPE EDIT LIMITCHARS MYPHONEP ID PHONE)))
```

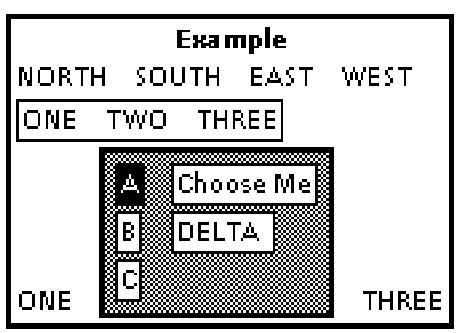
This menu has two command buttons, LOOKUP and EXIT, and three edit fields, with ID's NAME, PHONE, and ADDRESS. The Edit items are initialized to the empty string, as in this example they need no other initial value. The user could click after the Name: prompt, type a person's name, and then press the LOOKUP button. This would cause the function MYLOOKUPFN to be called, which could look at the NAME Edit item, lookup that name in the data base, and then fill in the rest of the fields appropriately. Note that the PHONE item has MYPHONEP as a LIMITCHARS function. This function would be called when editing the phone number, in order to restrict input to a valid phone number. After looking up Perry, the Free Menu might look like:



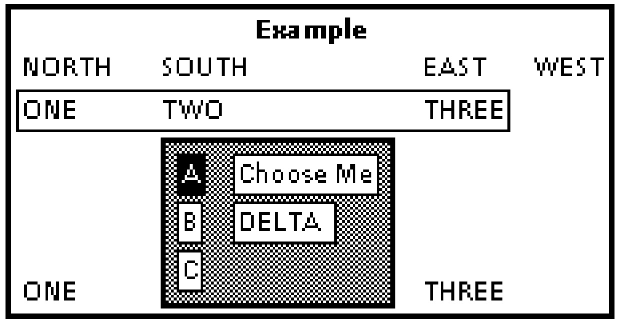
Here is a more complicated example:

```
((PROPS FONT (MODERN 10))
 ((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))
 ((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))
 ((PROPS ID ROW3 BOX 1)
  (LABEL ONE) (LABEL TWO) (LABEL THREE))
 ((PROPS ID ROW4)
  (LABEL ONE ID ALPHA)
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
         ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
          (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
          (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
         ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
           INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
          (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
   (LABEL THREE)))
```

which will produce the following Free Menu:



And if the Free Menu were formatted as a Table, instead of in Rows, it would look like:



Free Menu Group Properties

Each group has properties. Most group properties are relevant, and should be set, in the group's PROPS list in the Free Menu Description. User properties can freely be included in the PROPS list. A few other properties are setup by the formatter. After the Free Menu is created, group properties can be accessed by the macro FM.GROUPPROP or FM.MENUPROP.

ID

The identifier of this group. Setting the group ID is desirable, for example, if the application needs to get handles on items in particular groups, or access group properties.

FORMAT

One of ROW, COLUMN, TABLE, or EXPLICIT. The default is ROW.

FONT

A font description of the form (FAMILY SIZE FACE), or a FONTDESCRIPTOR data type. This will be the default font for each item in this group. The default font of the top group is the value of the variable DEFAULTFONT.

COORDINATES

One of GROUP, or MENU. This property applies only to Explicit formatting. If GROUP, then the items in the explicit group are positioned in coordinates reletive to the lower left corner of the group, as determined by the mother group. If MENU, which is the default, then the items are positioned reletive to the lower left corner of the menu.

LEFT

Specifies a left offset for this group, pushing the group to the right.

BOTTOM

Specifies a bottom offset for this group, pushing the group up.

ROWSPACE

The number of bits between rows in this group.

COLUMNSPACE

The number of bits between columns in this group.

BOX

The number of bits in the box around this group of items.

BOXSHADE

The shade of the box.

BOXSPACE

The number of bits between the box and the items.

BACKGROUND

The background shade of this group. Nested groups will inherit this background shade, but items in this group and nested groups will not. This is because in general it is difficult to read text on a background, so items appear on white background by default. This can be overridden by the BACKGROUND Item Property.

Other Group Properties

The following group properties are setup and maintained by Free Menu. The application should probably not change any of these properties.

ITEMS A list of the items in the group.

REGION The region that is the extent of the items in the group.

MOTHER The ID of the group that is the mother of this group.

DAUGHTERS A list of ID of groups which are daughters to this group.

Free Menu Items

Each Free Menu Item is stored as an instance of the Data Type FREEMENUITEM. Free Menu Items can be thought of as objects, each item having its own particular properties, such as its type, label, and mouse event functions. A number of useful item types, described in the section **Free Menu Item Types**, are predefined by Free Menu. New types of items can be defined by the application, using Display items as a base.

Each Free Menu Item is created from a Free Menu Item Description when the Free Menu is created.

Free Menu Item Descriptions

A Free Menu Item Description is a list in property list format, specifying the properties of the item. For example:

```
(LABEL Refetch SELECTEDFN MY.REFETCHFN)
```

describes a command (Momentary) item labelled 'Refetch', with the function MY.REFETCHFN to be called when the item is selected.

None of the property values in an item description are evaluated. When constructing Free Menu descriptions that incorporate evaluated expressions, for example labels that are bitmaps, it is helpful to use the backquote facility. For example, if the value of the variable MYBITMAP is a bitmap, then

```
(FREEMENU `(((LABEL A) (LABEL ,MYBITMAP))))
```

would create a Free Menu of one row, with two items in that row, the second of which has the value of MYBITMAP as its label.

Free Menu Item Properties

The following Free Menu Item Properties can be set in the Item Description. Any other properties given in an Item Description will be treated as user properties, and will be saved on the USERDATA property of the item.

TYPE The type of the item. Choose from one of the Free Menu Item type keywords MOMENTARY, TOGGLE, 3STATE, STATE, NWAY, EDITSTART, EDIT, NUMBER, or DISPLAY. The default is MOMENTARY.

LABEL	An atom, string, or bit map. Bit maps are always copied, so that the original won't be changed. This property must be specified for every item.
FONT	The font that the item will appear in. The default is the font specified for the group that this item is in. Can be a font description of the form (FAMILY SIZE FACE), or a FONTDESCRIPTOR data type.
ID	May be used to specify a unique identifier for this item, but is not necessary.
LEFT and BOTTOM	When Row, Column, or Table formatting, these specify offsets, pushing the item right and up, respectively, from where the formatter would have put the item. In Explicit formatting, these are the actual coordinates of the item, in the coordinate system given by the group's COORDINATES property.
HJUSTIFY	One of LEFT, CENTER, or RIGHT. Specifies that this item is to be horizontally justified within the extent of its group. Note that the main group, as opposed to the smaller row or column group, is used.
VJUSTIFY	One of TOP, MIDDLE, or BOTTOM. Specifies that this item is to be vertically justified.
HIGHLIGHT	Specifies the highlighted looks of the item, that is, how the item changes when a mouse event occurs on it. See the section **Free Menu Item Highlighting**, below.
MESSAGE	A string that will be printed in the prompt window after a mouse button is held down over this item for MENUHELDWAIT milliseconds. Or, if an atom, treat as a function to get the message. The function is applied to ITEM WINDOW BUTTONS, and should return a string. The default is a message appropriate to the type of the item.
INITSTATE	The initial state of the item. This is only appropriate to TOGGLE, 3STATE, and STATE items.
MAXWIDTH	The width allowed for this item. The formatter will leave enough space after the item for the item to grow to this width without collisions.
MAXHEIGHT	Similar to MAXWIDTH, but in the vertical dimension.
BOX	The number of bits in the box around this item. Boxes are made around MAXWIDTH and MAXHEIGHT dimensions. If unspecified, no box is drawn.
BOXSHADE	The shade that the box is drawn in. The default is BLACKSHADE.
BOXSPACE	The number of bits between the box and the label. The default is one bit.
BACKGROUND	The background shade on which the item appears. The default is WHITESHADE, regardless of the group's background.
LINKS	Can be used to link this item to other items in the Free Menu. See the section **Free Menu Item Links**.

Mouse Properties

The following properties provide a way for application functions to be called under certain mouse events. These functions are called with the ITEM, the WINDOW, and the BUTTONS depressed as arguments. These application functions do not interfere with any Free Menu system functions that take care of handling the different item types. In each case, though, the application function is called *after* the system function. The default for all of these functions is NILL. The value of each of the following properties can be the name of a function, or a lambda expression.

SELECTEDFN — The function to be called when this item is selected. Note that Edit and EditStart items cannot have a selectedfn. See Edit items, below.

DOWNFN — The function to be called when a mouse button goes down over this item, or when the mouse moves over the item with buttons depressed.

HELDFN — The function to be called repeatedly while the mouse is held down over this item.

MOVEDFN — The function to be called when the mouse moves off this item with buttons still depressed.

System Properties

The following Free Menu Item properties are set and maintained by Free Menu. The application should probably not change these properties directly.

GROUPID — The ID of the smallest group that the item is in. For example, in a row formatted group, the item's GROUPID will be set to the ID of the row that the item is in, not the ID of the whole group.

STATE — The current state of TOGGLE, 3STATE, or STATE items. The state of an NWAY item behaves like that of a toggle item.

BITMAP — The bitmap from which the item is displayed.

REGION — The region of the item, in window coordinates. This is used for locating the display position, as well as determing the mouse sensitive region of the item.

MAXREGION — The maximum region the item may occupy, determined by the MAXWIDTH and MAXHEIGHT properties. This is used by the formatter and the display routines.

SYSDOWNFN
SYSMOVEDFN
SYSSELECTEDFN — These are the system mouse event functions, setup by Free Menu according to the type of the item. These functions are called before the users mouse event functions, and are used to implement highlighting, state changes, editing, etc.

USERDATA — Any other properties are stored on this list in property list format. This list should probably not need to be manipulated directly.

Predefined Item Types

Momentary

Momentary items are like command buttons. When the button is selected, its associated function is called.

Toggle

Toggle items are simple two-state buttons. When depressed the button is highlighted, and it stays that way until pressed again. The states of a toggle button are T and NIL, initially NIL.

3State

3State items rotate through NIL, T, and OFF, states each time they are pressed. The default looks of the OFF state are with a diagonal line through the button, while T is highlighted, and NIL is normal. The default initial state is NIL.

The following Item Property applies to 3State items:

OFF — Specifies the looks of a 3STATE item in its OFF state. Similar to HIGHLIGHT. The default is that the label gets a diagonal slash through it.

State

State items are general multiple state items. The following Item Property determines how the item changes state:

CHANGESTATE — This Item Property can be changed at any time to change the effect of the item. If a MENU datatype, then this menu is popped up when the item is selected, and the user can select the new state. Otherwise, if this property is given, it is treated as a function name, which is applied to ITEM WINDOW BUTTONS. This function can do whatever it wants, and is expected to return the new state (an atom, string, or bitmap), or NIL, meaning don’t change state.

The state of the item can automatically be indicated in the Free Menu, by setting up a DISPLAY link to a Display item in the menu (see **Free Menu Item Links** below). If such a link exists, the label of the DISPLAY item will be changed to the new state. Note that the possible states are not restricted at all, except that if a popup menu is used, of course the possible selections are restricted. The state can be changed to any atom, string, or bitmap, manually via FM.CHANGESTATE.

The following Item Properties are relevent to State items when building a Free Menu:

MENUITEMS — If specified, should be a list of item to go in a popup menu for this item. Free Menu will build the menu and save it as the CHANGESTATE property of the item.

MENUFONT — The font of the items in the popup menu.

MENUTITLE — The title of the popup menu. The default title is the label of the State item.

Nway

NWay items provide a way to collect any number of items together, in any format within the Free Menu. Only one item from each Collection can be selected at a time, and that item is highlighted to indicate so.

The following Item Properties are particular to NWay items:

COLLECTION — An identifier that specifies which NWay Collection this item belongs to.

NWAYPROPS — A property list of information to be associated with this collection. This property is only noticed in the Free Menu Description on the first item in a Collection.

NWay Collections are formed by creating a number of NWay items with the same COLLECTION property. Each NWay item acts individually as a Toggle item, and can have its own mouse event functions.

Each NWay Collection itself has properties, its state for instance. After the Free Menu is created, these Collection properties can be accessed by the macro FM.NWAYPROPS. Note that NWay Collections are different from Free Menu Groups.

There are three NWay Collection properties that Free Menu looks at:

DESELECT
: If given, specifies that the Collection can be deselected, yielding a state in which no item in the Collection is selected. When this property is set, the Collection can be deselected by pressing the Right mouse button on any item in the Collection.

STATE
: The current state of the Collection, which is the actual item selected.

INITSTATE
: Specifies the initial state of the Collection. The value of this property is an item Link Description (see the section **Free Menu Item Links**.)

Edit

Edit items are textual items that can be edited. The label for an Edit item cannot be a bitmap. When the item is selected an edit caret appears at that cursor position within the item, allowing inserting and deleting characters at that point. If selected with the Right mouse button, the item is cleared before editing starts. While editing, the Left mouse button moves the caret to a new position within the item. The Right mouse button deletes from the caret to the cursor. Control-W deletes the previous word.

Editing is stopped when another item is selected, when the user clicks in another tty window, or by the Free Menu function FM.ENDEDIT, which is called when the Free Menu is reset, or the window is closed. Additionally, the Free Menu editor will time out after about a minute, returning automatically. Because of the many ways in which editing can terminate, Edit items are not allowed to have a Selectedfn, as it is not clear when this function should be called.

Each Edit item should have an ID specified, which is used when getting the state of the Free Menu, since the string being edited is defined as the state of the item, and thus cannot distinguish edit items. The following Item Properties are particular to Edit items:

MAXWIDTH
: Specifies the maximum string width of the item, in bits, after which input will be ignored. If MAXWIDTH is not specified, the items becomes ‘‘infinitely wide‘‘ and input is never restricted.

INFINITEWIDTH
: This property is set automatically when MAXWIDTH is not specified. This tells Free Menu that the item has no right end, so that the item becomes mouse sensitive from its left edge to the right edge of the window, within the vertical space of the item.

LIMITCHARS
: The input characters allowed can be restricted in two ways: If this item property is a list, it is treated as a list of legal characters; any character not in the list will be ignored. If it is an atom, it is treated as the name of a test predicate, which is applied to ITEM WINDOW CHARACTER when each character is typed. This predicate should return T if the character is legal, NIL otherwise. The LIMITCHARS function can also call FM.ENDEDIT to force the editor to terminate, or FM.SKIPNEXT, to cause the editor to jump to the next edit item in the menu.

ECHOCHAR
: This item property can be set to any character. This character will be echoed in the window, regardless of what character is typed. However the item’s label contains the actual string typed. This is useful for operations like password prompting. If ECHOCHAR is used, the font of the item must be fixed pitch.

Unrestricted Edit items should not have other items to their right in the menu, as they will be edited over. If the item is boxed, input is restricted to what will fit in the box. Typing off the edge of the window will cause the window to scroll appropriately. Control characters can be edited, including CR and LF, and they are echoed as a black box. While editing, the Skip/Next key ends editing the current item, and starts editing the next Edit item in the Free Menu.

Number

Number items are Edit items that are restricted to numerals. The state of the item is coerced to the the number itself, not a string of numerals.

There is one Number specific Item Property:

NUMBERTYPE — If FLOATP (or FLOAT), then decimals are accepted. Otherwise only whole numbers can be edited.

EditStart

EditStart items serve the purpose of starting editing on another item when they are selected. The associated Edit item is linked to the EditStart item by an EDIT link (see **Free Menu Item Links** below). If the EditStart item is selected with the Right mouse button, the Edit item is cleared before editing is started. Similar to Edit items, EditStart items cannot have a Selectedfn, as it is not clear when the associated editing will terminate.

Display

Display items serve two purposes. First, they simply provide a way of putting dummy text in a Free Menu, which does nothing when selected. The item's label can be changed, though. Secondly, Display items can be used as the base for new item types. The application can create new item types by specifying DOWNFN, HELDFN, MOVEDFN, and SELECTEDFN for a Display item, making it behave as desired.

Free Menu Item Highlighting

Each Free Menu Item can specify how it wants to be highlighted. First of all, if the item doesn't specify a HIGHLIGHT property, there are two default highlights. If the item is not boxed, the label is simply inverted, as in normal menus. If the item is boxed, it is highlighted in the shade of the box.

Alternatively, the value of the HIGHLIGHT property can be a SHADE, which will be painted on top of the item when a mouse event occurs on it. Or the HIGHLIGHT property can be an alternate label, which can be an atom, string, or bitmap. If the highlight label is a different size than the item label, the formatter will leave enough space for the larger of the two.

In all of these cases, the looks of the highlighted item are determined when the Free Menu is built, and a bitmap of the item with these looks is created. This bitmap is stored on the item's HIGHLIGHT property, and simply displayed when a mouse event occurs. The value of the highlight property in the Item Description is copied to the userdata list, in case it is needed later for a label change.

Free Menu Item Links

Links between items are useful for grouping items in abstract ways. In particular, links are used for associating Editstart items with their item to edit, and State items with their state display. The Free Menu Item property LINKS is a property list, where the value of each Link Name property is a pointer to another item.

In the Item Description, the value of the LINK property should be a property list as above. The value of each Link Name property is a Link Description.

A Link Descriptions can be one of the following forms:

`<ID>` Simply an ID of an item in the Free Menu. This is okay if items can be distinguished by ID alone.

`(<GROUPID> <ID>)` A list whose first element is a GROUPID, and whose second element is the ID of an item in that group. This way items with similar purposes, and thus similar ID's, can be distinguished across groups.

`(GROUP <ID>)` A list whose first element is the keyword GROUP, and whose second element is an item ID. This form describes an item with ID, in the same group that this item is in. This way you don't need to know the GROUPID, just which group you're in.

Then after the entire menu is built, the links are setup, turning the Link Descriptions into actual pointers to Free Menu Items. There is no reason why circular Item Links cannot be created, although such a link would probably not be very useful. If circular links are created, the Free Menu will not be garbage collected after it is not longer being used. The application is responsible for breaking any such links that it creates.

Free Menu Window Properties

FM.PROMPTWINDOW Specifies the window that Free Menu should use for displaying the item's messages. If not specified, PROMPTWINDOW is used.

FM.BACKGROUND The background shade of the entire Free Menu. This property can be set automatically by specifying a BACKGROUND argument to the function FREEMENU. The window border must be 4 or greater when a Free Menu background is used, due to the way the Window System handles window borders.

FM.DONTRESHAPE Normally Free Menu will attempt to use empty space in a window by pushing items around to fill the space. When a Free Menu window is reshaped, the items are repositioned in the new shape. This can be disabled by setting the FM.DONTRESHAPE window property.

Free Menu Interface Functions

(FREEMENU *DESCRIPTION TITLE BACKGROUND BORDER*) [Function]

Creates a Free Menu from a Free Menu Description, returning the window. This function will return quickly unless new display fonts have to be created. See the example above.

Accessing Macros

These Accessing Macros are provided to allow the application to get and set information in the Free Menu data structures. They are implemented as macros so that the operation will compile into the actual access form, rather than figuring that out at run time.

(FM.ITEMPROP *ITEM PROP {VALUE}*) [Macro]

Similar to WINDOWPROP, this macro provides an easy access to the fields of a Free Menu Item. A handle on the item can be gotten from the Free Menu by the function FM.GETITEM, described below. *VALUE* is optional, and if not given, the current value of the *PROP* property will be returned. If *VALUE* is given, it will be used as the new value for that *PROP*, and the old value will be returned.

When a call to FM.ITEMPROP is compiled, if the *PROP* is known (quoted in the calling form), the macro figures out what field to access, and the appropriate Data Type access form is compiled. However, if the *PROP* is not known at compile time, the *function* FM.ITEMPROP, which goes through the necessary property selection at run time, is compiled.

The TYPE and USERDATA properties of a Free Menu Item are Read Only, and an error will result from trying to change the value of one of these properties.

(FM.GROUPPROP *WINDOW GROUP PROP {VALUE}*) [Macro]

Provides access to the Group Properties set up in the PROPS list for each group in the Free Menu Description. *GROUP* specifies the ID of the desired group, and *PROP* the name of the desired property. If *VALUE* is specified, it will become the new value of the property, and the old value will be returned. Otherwise, the current value is returned.

(FM.MENUPROP *WINDOW PROP {VALUE}*) [Macro]

Provides access to the group properties of the top-most group in the Free Menu, that is to say, the entire menu. This provides an easy way for the application to attach properties to the menu as a whole, as well as access the Group Properties for the entire menu.

(FM.NWAYPROP *WINDOW COLLECTION PROP {VALUE}*) [Macro]

This macro works just like FM.GROUPPROP, except it provides access to the NWay Collections.

Accessing Functions

(FM.GETITEM *ID GROUP WINDOW*) [Function]

Get a handle on item *ID* in *GROUP* of the Free Menu in *WINDOW*. This function will search the Free Menu for an item whose ID property matches, or secondly whose LABEL property matches *ID*. If *GROUP* is NIL, then the entire Free Menu is searched. If no matching item is found, NIL is returned.

(FM.GETSTATE *WINDOW*) [Function]

Return in property list format the ID and current STATE of every NWay Collection and item in the Free Menu. If an item’s or Collection’s state is NIL, then it is not included in the list. This provides an easy way of getting the state of the menu all at once. If the state of only one item or Collection is needed, the application can directly access the STATE property of that object using the Accessing Macros above. Note that this function can be called when editing is in progress, in which case it will provide the label of the item being edited at that point.

Changing Free Menus

Many of the following functions operate on Free Menu Items, and thus take the item as an argument. The *ITEM* argument to these functions can be the Free Menu Item itself, or just a reference to the item. In the second case, FM.GETITEM will be used to find the item in the Free Menu.

The reference can be in one of the following forms:

`<ID>` Specifies the first item in the Free Menu whose ID or LABEL property matches <ID>.

`(<GROUPID> <ID>)` Specifies the item whose ID or LABEL property matches <ID> within the group specified by <GROUPID>.

(FM.CHANGELABEL *ITEM NEWLABEL WINDOW UPDATEFLG*) [Function]

This function changes an item's label after the Free Menu has been created. It works for any type of item, and state items will remain in their current state. If the window is open, the item will be redisplayed with its new appearance. *NEWLABEL* can be an atom, a string, or a bit map (except for Edit items), and will be restricted in size by the MAXWIDTH and MAXHEIGHT Item Properties. If these properties are unspecified, the item will be able to grow to any size. *UPDATEFLG* specifies whether or not the regions of the groups in the menu are recalculated to take into account the change of size of this item. The application should not change the label of an Edit item while it is being edited.

The following Item Property is relevant to changing labels:

CHANGELABELUPDATE Exactly like *UPDATEFLG* except specified on the item, rather than as a function paramater.

(FM.CHANGESTATE *X NEWSTATE WINDOW*) [Function]

Programmatically changes the state of items and NWay Collections. *X* is either an item or a Collection name. For items *NEWSTATE* is a state appropriate to the type of the item. For NWay Collections, *NEWSTATE* should be the desired item in the Collection, or NIL to deselect. For Edit and Number items, this function just does a label change. If the window is open, the item will be redisplayed.

(FM.RESETSTATE *ITEM WINDOW*) [Function]

Set an item back to its initial state.

(FM.RESETMENU *WINDOW*) [Function]

Reset every item in the menu back to its initial state.

(FM.RESETSHAPE *WINDOW ALWAYSFLG*) [Function]

Reshapes the window to its full extent, leaving the lower-left corner unmoved. Unless *ALWAYSFLG* is T, the window will only be increased in size as a result of resetting the shape.

(FM.RESETGROUPS *WINDOW*) [Function]

Recalculate the extent of each group in the menu, updating group boxes and backgrounds appropriately.

(FM.HIGHLIGHTITEM *ITEM WINDOW*) [Function]

This function provides a way of programmatically forcing an item to be highlighted. This might be useful for items which have a direct effect on other items in the menu. The item will be highlighted according to its HIGHLIGHT property, as described in the section **Free Menu Item Highlighting**. Note that this highlight is temporary, and will be lost if the item is redisplayed, by scrolling for example.

Editor functions

(FM.EDITITEM *ITEM WINDOW CLEARFLG*) [Function]

Start editing an Edit or Number item at the beginning of the item, as long as the window is open. This function will most likely be useful for starting editing of an item that is currently the null string. If *CLEARFLG* is set, the item is cleared first.

(FM.SKIPNEXT *WINDOW CLEARFLG*) [Function]

This function causes the editor to jump to the beginning of the next Edit item in the Free Menu. If *CLEARFLG* is set, then the next item will be cleared first. If there is not another Edit item in the menu, this function will simply cause editing to stop. If this function is called when editing is not in progress, editing will begin on the first Edit item in the menu. This function can be called from any process, and can also be called from inside the editor, in a LIMITCHARS function.

(FM.ENDEDIT *WINDOW WAITFLG*) [Function]

Stop any editing going on in *WINDOW*. If *WAITFLG*, then block until the editor has completely finished. This function can be called from another process, or from a LIMITCHARS function.

(FM.EDITP *WINDOW*) [Function]

If an item is in the process of being edited in the Free Menu *WINDOW*, that item is returned. Otherwise, NIL is returned.

Miscelaneous

(FM.REDISPLAYMENU *WINDOW*) [Function]

Redisplays the entire Free Menu in its window, if the window is open.

(FM.REDISPLAYITEM *ITEM WINDOW*) [Function]

Redisplays a particular Free Menu Item in its window, if the window is open.

(FM.SHADE *X SHADE WINDOW*) [Function]

X can be an item, or a group ID. *SHADE* is painted on top of the item or group. Note that this is a temporary operation, and will be undone by redisplaying. For more permanent shading, the application may be able to add a REDEDISPLAYFN and SCROLLFN for the window as necessary to update the shading.

(FM.WHICHITEM *WINDOW POSorX Y*) [Function]

Gets a handle on an item from its known location within the window. If *WINDOW* is NIL, (WHICHW) is used, and if *POSorX* is NIL, the current cursor location is used.

(FM.TOPGROUPID *WINDOW*) [Function]

Return the ID of the top group of this Free Menu.

Free Menu Changes:

This document describes the incompatible changes from the old version to the new version of Free Menu. This document does not describe any of the new features of Free Menu. Some of the terminology used in these notes is introduced in the Free Menu documentation. You should read the Free Menu documentation first.

The function FREEMENU is used to create a Free Menu, replacing and combining the functions FM.MAKEMENU and FM.FORMATMENU.

In the Description of the Free Menu:

There is no longer a WINDOWPROPS list in the Free Menu Description. Instead, the window properties TITLE and BORDER that used to be set in the WINDOWPROPS list can now be passed to the function FREEMENU. Other window properties (like FM.PROMPTWINDOW) can be set directly after FREEMENU returns the window using the system function WINDOWPROP. See the section in the documentation entitled **Free Menu Window Properties**.

Setting the initial state of an item is now done with the item property INITSTATE in the item description, rather than the STATE property.

Free Menu Items:

3STATE items now have states OFF, NIL, and T (instead of a NEUTRAL state). They appear by default in the NIL state.

STATE items are general purpose items which maintain state, and replace the functionality of NCHOOSE items. To get the functionality of NCHOOSE items, specify the property MENUITEMS (a list of items to go in a popup menu), which instructs the STATE item to popup the menu when it is selected. STATE items do not display their current state by default, like NCHOOSE items used to. Instead, if you want the state displayed in the Free Menu, you have to link the STATE item to a DISPLAY item using a Free Menu Item Link named "DISPLAY". The current state of the STATE item will then automatically be displayed in the specified DISPLAY item. The item properties MENUFONT and MENUTITLE also apply to the popup menu.

NWAY items are declared slightly differently. There is now the notion of an NWay Collection, which is a collection of items acting an a single nway item. The Collection is declared by specifying any number of NWay items, each with the same COLLECTION property. NWay Collections have properties themselves, accessible by the macro FM.NWAYPROPS. These properties can be specified in property list format as the value of the NWAYPROPS Item Property of the first NWay item declared for each Collection. NWay Collections by default cannot be deselected (a state in which no item selected). Setting the Collection property DESELECT to any non-nil value changes this behavior. The state of the NWay Collection is maintained in its STATE property.

EDIT items no longer will stop at the edge of the window. Editing is either restricted by the MAXWIDTH property, or else it is not restricted at all. The EDITSTOP property is obsolete. Starting editing with the Right mouse button causes the item to be cleared first.

EDITSTART items now specify their associated edit item (there can only be one, now) by a Free Menu Item Link named "EDIT" from the EDITSTART item to the EDIT item.

TITLE items are replaced by DISPLAY items, which work the same way.

Free Menu Interface functions:

(FREEMENU DESCRIPTION TITLE BACKGROUND BORDER) replaces FM.MAKEMENU and FM.FORMATMENU.
The desired format is not specified as the value of the FORMAT property in the group's PROPS list.

(FM.GETITEM ID GROUP WINDOW) replaces FM.ITEMFROMID.
Searches within GROUP for an item whose ID property is *ID*. *ID* is matched against the item ID and then the item LABEL. If *GROUP* is NIL, the entire menu is searched.

(FM.GETSTATE WINDOW) replaces FM.READSTATE.
Returns a property list of the selected item in the menu. This list now also includes the NWay Collections and their selected item.

(FM.CHANGELABEL ITEM NEWLABEL WINDOW UPDATEFLG) **new argument order**.
Now works by rebuilding the item label from scratch, taking the original specification of MAXWIDTH and MAXHEIGHT into account. NEWLABEL can be an atom, string, or bitmap. If UPDATEFLG is set, then the Free Menu Group's regions are recalculated, so that boxed groups will be redisplayed properly.

(FM.CHANGESTATE X NEWSTATE WINDOW) **new argument order**.
X is either an item or an NWay Collection ID. NEWSTATE is an appropriate state to the type of item. If an NWay collection, NEWSTATE is the actual item to be selected, or NIL to deselect. Toggle items take either T or NIL as NEWSTATE, and 3STATE items take OFF, NIL, or T, and STATE items take any atom, string, or

bitmap as their new state. For EDIT items, *NEWSTATE* is the new label, and FM.CHANGELABEL is called to change the label of the EDIT item.

(FM.RESETSHAPE WINDOW ALWAYSFLG) replaces FM.FIXSHAPE

(FM.HIGHLIGHTITEM ITEM WINDOW) replaces FM.SHADEITEM and FM.SHADEITEMBM.
FM.HIGHLIGHTITEM will programmatically highlight an item, as specified by its HIGHLGIHT property. The highlighting is temporary, and will be undone by a redisplay or scroll. To programmatically shade an item an arbitrary shade, use the new function FM.SHADE.

ICONW

This is a package of functions for building small windows of arbitrary shape, principally for use as icons for shrinking windows; i.e., these functions are likely to be invoked from within the ICONFN of a window.

An icon is specified by supplying its image (a bit map) and a mask that specifies its shape. The mask is a bit map of the same dimensions as the image whose bits are on (black) in those positions considered to be in the image, off (white) in those positions where the background should ‘‘show through.’’ By using the mask and appropriate window functions, the icon package maintains the illusion that the icon window is nonrectangular, even though the actual window itself is rectangular. The illusion is not complete, of course. For example, if you try to select what looks like the background (or an occluded window) around the icon but still within its rectangular perimeter, the icon window itself is selected. Also, if you move a window occluded by an icon, the icon never notices that the background changed behind it.

Icons created with this package can also have ‘‘titles’’; some part of the image can be filled with text computed at the time the icon is created, or even changed after creation.

Creating Icons

(ICONW *IMAGE MASK POSITION NOOPENFLG*) [Function]

Creates a window at *POSITION*, or prompts for a position if *POSITION* is NIL. The window is borderless, and filled with *IMAGE*, as cookie-cut by *MASK*. If *MASK* is NIL, the image is considered rectangular (i.e., *MASK* defaults to a black bit map of the same dimensions as *IMAGE*). If *NOOPENFLG* is T, the window is returned unopened.

(TITLEDICONW *ICON TITLE FONT POSITION NOOPENFLG JUST BREAKCHARS OPERATION*) [Function]

Creates a titled icon at *POSITION*, or prompts for a position if *POSITION* is NIL. If *NOOPENFLG* is T, the window is returned unopened. The argument *ICON* is an instance of the record TITLEDICON, which specifies the icon image and mask, as with ICONW, and a region within the image to be used for displaying the title. Thus, the *ICON* argument is usually of the form

(create TITLEDICON ICON_*someIconImage*

MASK_*iconMask* TITLEREG_*someRegionWithinICON*).

The title region is specified in icon-relative coordinates, i.e., the lower-left corner of the image bit map is (0, 0). The mask can be NIL if the icon is rectangular. The image should be white where it is covered by the title region (in any event, TITLEDICONW clears the region before printing on it).

The title is printed into the specified region in the image, using *FONT*, which if NIL defaults to the value of DEFAULTICONFONT, initially Helvetica 10. The title is broken into multiple lines if necessary; TITLEDICONW attempts to place the breaks at characters that are in the list of character codes *BREAKCHARS*. *BREAKCHARS* defaults to (CHARCODE (SPACE ÿ)). In addition, line breaks are forced by any carriage returns in *TITLE*, independent of *BREAKCHARS*. *BREAKCHARS* is ignored as needed if a long title would not otherwise fit in the specified region. For convenience, *BREAKCHARS* = FILE means the title is a file name, so break at file name field delimiters.

The argument *JUST* indicates how the text should be justified relative to the region—it is an atom or list of atoms chosen from TOP, BOTTOM, LEFT, or RIGHT, which indicate the vertical positioning (flush to top or bottom) and/or horizontal positioning (flush to left edge or right). Where not indicated, the text is centered.

The argument *OPERATION* is a display stream operation indicating how the title should be printed. If *OPERATION* is INVERT, then the title is printed white-on-black. The default *OPERATION* is REPLACE, meaning black-on-white. ERASE is the same as INVERT; PAINT is the same as REPLACE.

For convenience, TITLEDICONW can also be used to create icons that consist solely of a title, with no special image. If the argument *ICON* is NIL, TITLEDICONW creates a rectangular icon large enough to contain *TITLE*, with a border the same width as a regular window. The remaining arguments are as described above, except that a *JUST* of TOP or BOTTOM is not meaningful.

Modifying Icons

(ICONW.TITLE *ICON TITLE*) [Function]

Returns the current title of the window *ICON*, which must be a window returned by TITLEDICONW. Additionally, if *TITLE* is non-NIL, makes *TITLE* be the new title of the window and repaints it accordingly. To erase the current title, make *TITLE* be a null string.

(ICONW.SHADE *WINDOW SHADE*) [Function]

Returns the current shading of the window *ICON*, which must be a window returned by ICONW or TITLEDICONW. Additionally, if *SHADE* is non-NIL, paints the texture *SHADE* on *WINDOW*. A typical use for this function is to communicate a change of state in a window that is shrunk, without reopening the window. To remove any shading, make *SHADE* be WHITESHADE.

Default Icons

When you shrink a window that has no ICONFN, the system currently creates an icon that looks like the window's title bar. You can make the system instead create titled icons by setting the global variable DEFAULTICONFN to the value TEXTICON.

(TEXTICON *WINDOW TEXT*) [Function]

Creates a titled icon window for the main window *WINDOW* containing the text *TEXT*, or the window's title if *TEXT* is NIL.

DEFAULTTEXTICON [Variable]

The value that TEXTICON passes to TITLEDICONW as its ICON argument. Initially NIL, which creates an unadorned rectangular window, but you can set it to a TITLEDICON record of your choosing if you would like default icons to have a different appearance.

Sample Icons

The file <LispUsers>StockIcons contains a collection of icons and their masks usable with ICONW, including:

FOLDER, FOLDERMASKžA file folder

PAPERICON, PAPERICONMASK—A sheet of paper with the top right corner turned

FILEDRAWER, FILEDRAWERMASK—The front of a file drawer

ENVELOPEICON, ENVELOPEMASK—An envelope

TITLED.FILEDRAWER—A TitledIcon of the filedrawer front (Capacity, about three lines of 10-pt. text)

TITLED.FILEFOLDER—A TitledIcon of the file folder (Capacity, about three lines of 10-pt. text)

TITLED.ENVELOPE—A TitledIcon of the envelope (Capacity, one short line of 10-pt. text)

APPENDIX B. SEDIT—THE LISP EDITOR

SEdit is the Xerox Lisp structure editor. It allows you to edit Xerox Lisp code directly in memory. This editor replaces DEdit in Chapter 16, Structure Editor, of the *Interlisp-D Reference Manual*. First introduced in Lyric, the SEdit structure editor has been greatly enhanced in the Medley release. Medley additions are indicated with revision bars in the right margin.

All symbols referenced in this appendix are external in the package named "SEdit", unless otherwise qualified.

16.1 SEdit - The Structure Editor

As a structure editor, SEdit alters Lisp code directly in memory. The effect this has on the running system depends on what is being edited.

For Common Lisp definitions, SEdit always edits a copy of the object. For example, with functions, it edits the definition of the function. What the system actually runs is the installed function, either compiled or interpreted. The primary difference between the definition and the installed function is that comment forms are removed from the definition to produce the installed function. The changes made while editing a function will not be installed until the edit session is complete.

For Interlisp functions and macros, SEdit edits the actual structure that will be run. An exception to this is an edit of an EXPR definition of a compiled function. In this case, changes are included and the function is unsaved when the edit session is completed.

SEdit edits all other structures, such as variables and property lists, directly. SEdit installs all changes as they are made.

If an error is made during an SEdit session, abort the edit with an Abort command (see Section 16.1.7, Command Keys). This command undoes all changes from the beginning of the edit session and exits from SEdit without changing your environment.

If the definition being edited is redefined while the edit window is open, SEdit redisplays the new definition. Any edits on the old definition will be lost. If SEdit was busy when the redefinition occurred, the SEdit window will be gray. When SEdit is no longer busy, position the cursor in the SEdit window and press the left mouse button; SEdit will get the new definition and display it.

16.1.1 An Edit Session

The List Structure Editor discussion in Chapter 3, Language Integration, explains how to start an editor in Xerox Lisp.

Whenever you call SEdit, a new SEdit window is created. This SEdit window has its own process, and thus does not rely on an Exec to run in. You can make edits in the window, shrink it while you do something else, expand it and edit some more, and finally close the window when you are done.

Throughout an edit session, SEdit remembers everything that you do through a change history. All edits can be undone and redone sequentially. When an edit session ends, SEdit forgets this information and installs the changes in the system.

The session ends with an event signalling to the editor that changes are complete. Three events signal completion:

- Closing the window.

Do this to terminate the edit session when you are finished.

- Shrinking the window.

Shrink the window when you have made some edits and may want to continue the editing session at a later time.

- Typing one of the Completion Commands, listed below.

Each of these commands has the effect of installing your changes, completing the edit, and returning the TTY process to the Exec. They vary in what is done in addition to completing. Using these commands the definition that you were editing can be automatically compiled, the edit window can be closed, or both.

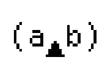
A new edit session begins when you come back to an SEdit after completing. The change history is discarded at this point.

If the Exec is waiting for SEdit to return before going on, complete the edit session using any of the methods above to alert the Exec that SEdit is done. The TTY process passes back to the Exec .

16.1.2 SEdit Carets

There are two carets in SEdit, the edit caret and the structure caret. The edit caret appears when characters are edited within a single structure, such as an atom, string, or comment. Anything typed in will appear at the edit caret as part of the structure that the caret is within. The edit caret looks like this:

The structure caret appears when the edit point is between structures, so that anything inserted will go into a new structure. It looks like this:



SEdit changes the caret frequently, depending on where you are in the structure you are editing, and how the caret is positioned. The left mouse button allows an edit caret position to be set. The middle mouse button allows the structure caret position to be set .

16.1.3 The Mouse

In SEdit, the mouse buttons are used as follows. The left mouse button positions the mouse cursor to point to parts of Lisp structures. The middle mouse button positions the mouse cursor to point to whole Lisp structures. Thus, selecting the Q in LEQ using the left mouse button selects that character, and sets the edit caret after the Q:

```
(LEQ n 1)
```

Any characters typed in at this point would be appended to the atom LEQ.

Selecting the same letter using the middle mouse button selects the whole atom (this convention matches TEdit's character/word selection convention), and sets a structure caret between the LEQ and the n:

```
(LEQ n 1)
```

At this point, any characters typed in would form a new atom between the LEQ and the n.

Larger structures can be selected in two ways. Use the middle mouse button to position the mouse cursor on the parenthesis of the desired list to select that list. Press the mouse button multiple times, without moving the mouse, extends the selection. Using the previous example, if the middle button were pressed twice, the list (LEQ ...) would be selected:

```
(LEQ n 1)
```

Pressing the button a third time would cause the list containing the (LEQ n 1) to be selected.

The right mouse button positions the mouse cursor for selecting sequences of structures or substructures. Extended selections are indicated by a box enclosing the structures selected. The selection is extended in the same mode as the original selection. That is, if the original selection were a character selection, the right button could be used to select more characters in the same atom. Extended selections also have the property of being marked for pending deletion. That is, the selection takes the place of the caret, and anything typed in is inserted in place of the selection.

For example, selecting the E by pressing the left mouse button and selecting the Q by pressing the right mouse button would produce:

```
(LEQ n 1)
```

Similarly, pressing the middle mouse button and then selecting with the right mouse button extends the selection by whole structures. Thus, in our example, pressing the middle mouse button to select LEQ and pressing the right mouse button to select the 1 would produce:

```
(LEQ n 1)
```

This is not the same as selecting the entire list, as above. Instead, the elements in the list are collectively selected, but the list itself is not.

16.1.4 Gaps

The SEdit structure editor requires that everything edited must have an underlying Lisp structure, even if the structure is not directly displayed. For example, with quoted forms the actual structure might be (**QUOTE** GREEN), although this would be displayed as `'GREEN`. Even when the user is in the midst of typing in a form, the underlying Lisp structure must exist.

Because of this necessity, SEdit provides gaps to serve as dummy Lisp objects during typing. SEdit does not need a gap for every form typed in, but gaps are necessary for quoted objects. When something is typed that requires SEdit to build a Lisp structure and thus create a gap, as the quote character does, the gap will appear marked for pending deletion. This means it is ready to be replaced by the structure to be typed in. In this way it is possible to type special structures, like quotes, directly, while SEdit maintains the structure.

A gap looks like: –x–

A gap displayed after a quote has been typed in would look like this:

'–x–

with the gap marked for pending deletion, ready for typein of the object to be quoted.

16.1.5 Special Characters

A few characters have special meaning in Lisp, and are treated specially by SEdit. SEdit must always have a complete structure to work on at any level of the edit. This means that SEdit needs a special way to type in structures such as lists, strings, and quoted objects. In most instances these structures can be typed in just as they would be to a regular Exec, but in a few cases this is not possible.

Lists- (and)

Lists begin with an open parenthesis character **(**. Typing an open parenthesis gives a balanced list, that is, SEdit inserts both an open and a close parenthesis. The structure caret is between the two parentheses. List elements can be typed in at the structure caret. When a close parenthesis, **)** is typed, the caret will be moved outside the list (and the close parenthesis), effectively finishing the list. Square bracket characters, [and], have no special meaning in SEdit, as they have no special meaning in Common Lisp.

Quoted Structures:

SEdit handles the quote keys so that it is possible to type in all quote forms directly. When typing one of the following quote keys at a structure caret, the quote character typed will appear, followed by a gap to be replaced by the object to be quoted.

Single Quote – ' Use to enter quoted structures.

Backquote –' Use to enter backquoted structures.

Comma – , Use to enter comma forms, as used with a Backquote form.

At Sign – @ Use after a comma to create a comma-at-sign gap. This allows type-in of comma-at forms, e.g. **,@list**, as used within a Backquote form.

Dot – . Use the dot (period) after a comma to create a comma-dot gap. This allows type-in of comma-dot forms, e.g. **,.list**, as used within a Backquote form.

Hash Quote – #' Use this two character sequence to enter the **CL:FUNCTION** abbreviation hash–quote (#').

Dotted Lists:

The dot, or period, character (**.**) is used to type dotted lists in SEdit. After typing a dot, SEdit inserts a dot and a gap to fill in for the tail of the list. To dot an existing list, point the cursor between the last and second to the last element in the list, and type a dot. To undot a list, select the tail of the list before the dot while holding down the SHIFT key.

Escape- \ or %

Use to escape from a specific typed in character. Use the escape key to enter characters, like parentheses, which otherwise have special meaning to the SEdit reader. Press the escape key then type in the character to escape. SEdit uses the escape key appropriate to the environment it is editing in; it depends on the readtable that was current when the editor was started. The backslash key (****) is used when editing Common Lisp, and the percent key (**%**) is used when editing Interlisp.

Multiple Escape- |

Use the multiple escape key, the vertical bar character (**|**), to escape a sequence of typed in characters. SEdit always balances multiple escape characters. When one multiple escape character is typed, SEdit produces a balanced pair, with the caret between them, ready for typing in the characters to be escaped. If you type a second vertical bar, the caret moves after the second vertical bar, and is still within the same atom, so that you can add more unescaped characters to the atom.

Comments- ;

The comment key, a semicolon (**;**), starts a comment. When a semicolon is typed, an empty comment is inserted with the caret in position for typing in the comment. Comments can be edited like strings. There are three levels of comments supported by SEdit: single, double, and triple. Single semicolon comments are formatted at the comment column, about three-quarters of the way across the SEdit window, towards the right margin. Double semicolon comments are formatted at the current indentation of the code that they are in. Triple semicolon comments are formatted against the left margin of the SEdit window. The level of a comment can be increased or decreased by pointing after the semicolon, and either typing another semicolon, or backspacing over the preceding semicolon. Comments can be placed anywhere in your Common Lisp code. However, in Interlisp code, they must follow the placement rules for Interlisp comments.

Strings- "

Enter strings in SEdit by typing a double quote (**"**). SEdit balances the double quotes. When one is typed, SEdit produces a second, with the caret between the two, ready for typing the characters of the string. If a double quote character is typed in the middle of a string, SEdit breaks the string into two smaller strings, leaving the caret between them.

16.1.6 Commands

SEdit commands are most easily entered through the keyboard. When possible, SEdit uses a named key on the keyboard, for example, the DELETE key. The other commands are either Meta, Control, or Meta-Contol key combinations. For the alphabetic command keys, either uppercase or lowercase will work.

There are two menus available, as an alternative means of invoking commands. They are the middle button popup menu, and the attached command menu. These menus are described in more detail below.

16.1.6 Editing Commands

Redisplay: Control-L [Editor Command]

Redisplays the structure being edited.

Delete Selection: DELETE [Editor Command]

Deletes the current selection.

Delete Word: Control-W [Editor Command]

Deletes the previous atom or whole structure. If the caret is in the middle of an atom, deletes backward to the beginning of the atom only.

16.1.7 Completion Commands

Abort: Meta-A [Editor Command]

Aborts. This command must be confirmed. All changes since the beginning of the edit session are undone, and the edit is closed.

The following commands signal completion of an edit session and install the structure you were editing.

Control-X [Editor Command]

Signals the system that this edit is complete. The window remains open, though, so the user can see the edit and start editing again directly.

Control-C [Editor Command]

Signals the system that this edit is complete and compiles the definition being edited. The variable *compile-fn* determines the function to be called to do the compilation. See the Options section below.

Meta-Control-X [Editor Command]

Signals the system that this edit is complete and closes the window.

Meta-Control-C [Editor Command]

Signals the system that this edit is complete, compiles the definition being editing, and closes the window.

16.1.8 Undo Commands

Undo: Meta-U or UNDO [Editor Command]

Undoes the last edit. All changes since the beginning of the edit session are remembered, and can be undone sequentially.

Redo: Meta-R or AGAIN [Editor Command]

Redoes the edit change that was just undone. Redo only works directly following an Undo. Any number of Undo commands can be sequentially redone.

16.1.9 Find Commands

Find: Meta-F or FIND [Editor Command]

Finds a specified structure, or sequence of structures. If there is a current selection, SEdit looks for the next occurrence of the selected structure. If there is no selection, SEdit prompts for the structure to find, and searches forward from the position of the caret. The found structure will be selected, so the Find command can be used to easily find the same structure again.

If a sequence of structures is selected, SEdit will look for the next occurrence of the same sequence. Similarly, when SEdit prompts for the structure to find, you can type a sequence of structures to look for.

The variable *wrap-search* controls whether or not SEdit wraps around from the end of the structure being edited and continues searching from the beginning.

Reverse Find: Control-Meta-F [Editor Command]

Finds a specified structure, searching in reverse from the position of the caret.

The variable *wrap-search* controls whether or not SEdit wraps around from the beginning of the structure being edited and continues searching from the end.

Find Gap: Meta-N or SKIP-NEXT [Editor Command]

Skips to the next gap in the structure, leaving it selected for pending deletion.

Substitute: Meta-S or SHIFT-FIND [Editor Command]

Substitutes one structure, or sequence of structures, for another structure, or sequence, within the current selection. SEdit prompts you in the SEdit prompt window for the structures to replace, and the structures to replace with.

The selection to substitute within must be a structure selection. To get a structure selection, click with the middle mouse button (not the left), and extend it, if necessary, with the right mouse button. If you begin with the left button, you will get an informational message "Select the structure to substitute within", because the selection was of characters, rather than structures.

Delete Structure: Meta-Control-S [Editor Command]

Removes all occurences of a structure or sequence of structures within the current selection. SEdit prompts the user in the SEdit prompt window for the structures to delete.

16.1.9 General Commands

Arglist: Meta-H or HELP [Editor Command]

Shows the argument list for the function currently selected, or currently being typed in, in the SEdit prompt window. If the argument list will not fit in the SEdit prompt window, it is displayed in the main Prompt Window.

Convert Comments: Meta-; [Editor Command]

Converts old style comments in the selected structure to new style comments. This converter notices any list that begins with an asterisk (*) in the INTERLISP package (IL:*) as an old style comment. Section 16.1.11, Options, describes the converter options .

Edit: Meta-O [Editor Command]

Edits the definition of the current selection. If the selected name has more than one type of definition, SEdit asks for the type to be edited. If the selection has no definition, a menu pops up. This menu lets the user specify either the type of definition to be created, or no definition if none needs to be created.

Eval: Meta-E or Do-It [Editor Command]

Evaluates the current selection. If the result is a structure, the inspector is called on it, allowing the user to choose how to look at the result. Otherwise, the result is printed in the SEdit prompt window. The evaluation is done in the process from which the edit session was started. Thus, while editing a function from a break window, evaluations are done in the context of the break.

Expand: Meta-X or EXPAND [Editor Command]

Replaces the current selection with its definition. This command can be used to expand macros and translate CLISP.

Extract: Meta- / [Editor Command]

Extracts one level of structure from the current selection. If the current selection is an atom, or if there is no selection, the next largest structure containing this atom, or caret, is used. This command can be used to strip the parentheses off a list or a comment, or to unquote a quoted structure.

Inspect: Meta-I [Editor Command]

Inspect the current selection.

Join: Meta-J [Editor Command]

Joins. This command joins any number of sequential Lisp objects of the same type into one object of that type. Join is supported for atoms, strings, lists, and comments. In addition, SEdit permits joining of a sequence of atoms and strings, since either type can easily be coerced into the other. In this case, the result of the Join will be an atom if the first object in the selection is an atom, otherwise the result will be a string.

Mutate: Meta-Z [Editor Command]

Mutates. This command allows the user to do arbitrary operations on a LISP structure. First select the structure to be mutated (it must be a whole structure, not an extended selection). When the user presses Meta-Z SEdit prompts for the function to use for mutating. This function is called with the selected structure as its argument, and the structure is replaced with the result of the mutation.

For example, an atom can be put in upper case by selecting the atom and mutating by the function U-CASE. You can replace a structure with its value by selecting it and mutating by EVAL.

Quote: Meta-'
Meta-‘
Meta-,
Meta-.
Meta-@ or Meta-2
Meta-# or Meta-3 [Editor Command]

Quotes the current selection with the specified kind of quote, respectively, Single Quote, Backquote, Comma, Comma-At-Sign, Comma-Dot, or Hash-Quote.

Parenthesize: Meta-) or Meta-0 [Editor Command]

Parenthesizes the current selection, positioning the caret after the new list.

Parenthesize: Meta- (or Meta-9 [Editor Command]

Parenthesizes the current selection, positioning the caret at the beginning of the new list. Only a whole structure selection or an extended selection of a sequence of whole structures can be parenthesized.

16.1.10 Miscellaneous

Change Print Base: Meta-B [Editor Command]

Changes Print Base. Prompts for entry of the desired Print Base, in decimal. SEdit redisplays fixed point numbers in this new base.

Set Package: Meta-P [Editor Command]

Changes the current package for this edit. Prompts the user, in the SEdit prompt window, for a new package name. SEdit will redisplay atoms with respect to that package.

Attached Menu: Meta-M [Editor Command]

Attaches a menu of the commonly used commands (the SEdit Command Menu) to the top of the SEdit window. Each SEdit window can have its own menu, if desired.

16.1.10 Help Menu

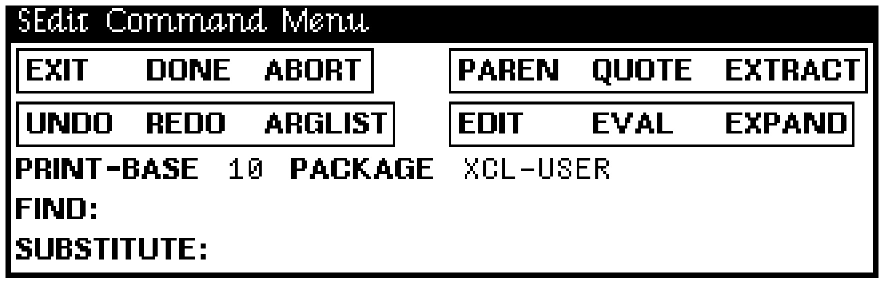
When the mouse cursor is positioned in the SEdit title bar and the middle mouse button is pressed, a Help Menu of commands pops up. The menu looks like this:

Commands	
Abort	M-A
Done	C-X
Done & Compile	C-C
Done & Close	M-C-X
Done, Compile, & Close	M-C-C
Undo	M-U
Redo	M-R
Find	M-F
Reverse Find	M-C-F
Remove	M-C-S
Substitute	M-S
Find Gap	M-N
Arglist	M-H
Convert Comment	M-;
Edit	M-O
Eval	M-E
Expand	M-X
Extract	M-/
Inspect	M-I
Join	M-J
Mutate	M-Z
Parenthesize	M-(
Quote	M-'
Set Print-Base	M-B
Set Package	M-P
Attach Menu	M-M

The Help Menu lists each command and its corresponding Command Key. (In the menu, the letter C stands for CONTROL, while M indicates Meta.) The command selected is executed just as if the command had been entered from the keyboard. The menu remembers which command was selected last, and pops up with the mouse cursor next to that same command the next time the menu is used. This provides a very fast way to repeat the same command when using the mouse.

16.1.9 Command Menu

The SEdit Attached Command Menu contains the commonly used commands. Use the Meta-M keyboard command to bring up this menu. The menu can be closed, independently of the SEdit window, when desired. The menu looks like:



All of the commands in the menu function identically to their corresponding keyboard commands, except for Find and Substitute.

When Find is selected with the mouse cursor, SEdit prompts in the menu window, next to the Find button, for the structures to find. Type in the structures then select Find again. The search begins from the caret position in the SEdit window.

Similarly, Substitute prompts, next to the Find button, for the structures to find, and next to the Substitute button for the structures to substitute with. After both have been typed in, selecting Substitute replaces all occurrences of the Find structures with the Substitute structures, within the current selection.

To do a confirmed substitute, set the edit point before the first desired substitution, and select Find. Then if you want to substitute that occurrence of the structure, select Substitute. Otherwise, select Find again to go on.

Selecting either Find or Substitute with the right mouse button erases the old structure to find or substitute from the menu, and prompts for a new one.

16.1.11 SEdit Window Region Manager

SEdit provides user redefinable functions which control how SEdit chooses the region for a new edit window.

(Get-Window-Region *CONTEXT REASON NAME TYPE***)** [Function**]**

This function is called when SEdit wants to know where to place a window it is about to open. This happens whenever the user starts a new SEdit or expands an Sedit icon.The default behavior is to pop a window region off SEdit's stack of regions that have been used in the past. If the stack is empty, SEdit prompts for a new region.

This function can be redefined to provide different behavior. It is called with the edit *CONTEXT*, a *REASON* for needing a region, the *NAME* of the structure to be edited, and the *TYPE* of the structure to be edited. The edit *CONTEXT* is SEdit's main data structure and can be useful for associating particular edits with specific regions. The *REASON* argument specifies why SEdit wants a region, and will be one of the keywords :CREATE or :EXPAND.

(Save-Window-Region *CONTEXT REASON NAME TYPE REGION***)** [Function]

This function is called whenever SEdit is finished with a region and wants to make the region available for other SEdits. This happens whenever an SEdit window is closed or shrunk, or when an SEdit Icon is closed. The default behavior is simply to push the region onto SEdit's stack of regions.

This function can be redefined to provide different behavior. It is also called with the edit *CONTEXT*, the *REASON*, the *NAME*, the *TYPE*, and additionally the window *REGION* that is being released. The *REASON* argument specifies why SEdit is releasing the region, and will be one of the keywords :CLOSE, :SHRINK, or :CLOSE-ICON.

Keep-Window-Region [Variable]

Default **T**. This flag determines the behavior of the default SEdit region manager, explained above, for shrinking and expanding windows. When set to **T**, shrinking an SEdit window will not give up that window's region; the icon will always expand back into the same region. When set to **NIL**, the window's region is made available for other SEdits when the window is shrunk. Then when an SEdit icon is expanded, the window will be reshaped to the next available region.

This variable is only used by the default implementations of the functions **Get-Window-Region** and **Save-Window-Region**. If these functions are redefined, this flag is no longer used.

16.1.12 Options

The following parameters can be set as desired.

Wrap-Parens [Variable]

This SEdit pretty printer flag determines whether or not trailing close parenthesis characters,), are forced to be visible in the window without scrolling. By default it is set to NIL, meaning that close parens are allowed to "fall off" the right edge of the window. If set to T, the pretty printer will start a new line before the structure preceding the close parens, so that all the parens will be visible.

Wrap-Search [Variable]

This flag determines whether or not SEdit find will wrap around to the top of the structure when it reaches the end, or vice versa in the case of reverse find. The default is NIL.

Clear-Linear-On-Completion [Variable]

This flag determines whether or not SEdit completely re-pretty prints the structure being edited when you complete the edit. The default value is NIL, meaning that SEdit reuses the pretty printing.

Convert-Upgrade [Variable]

Default 100. When using Meta-; to convert old-style single- asterisk comments, if the length of the comment exceeds **Convert-Upgrade** characters, the comment is converted into a double semicolon comment. Otherwise, the comment is converted into a single semicolon comment.

Old-style double-asterisk comments are always converted into new-style triple-semicolon comments.

16.1.13 Control Functions

(Reset) [Function**]**

This function recomputes the SEdit edit environment. Any changes made in the font profile, or any changes made to SEdit's commands are captured by resetting. Close all SEdit windows before calling this function.

(Add-Command *KEY-CODE FORM &OPTIONAL KEY-NAME COMMAND-STRING HELP-STRING***)** [Function**]**

This function allows you to write your own SEdit keyboard commands. You can add commands to new keys, or you can redefine keys that SEdit already uses as command keys. If you mistakenly redefine an SEdit command, the funtion Reset-Commands will remove all user-added commands, leaving SEdit with its default set of commands.

KEY-CODE can be a character code, or any form acceptible to il:charcode.

FORM determines the function to be called when the key command is typed. It can be a symbol naming a function, or a list, whose first element is a symbol naming a function and the rest of the elements are extra arguments to the function. When the command is invoked, SEdit will apply the function to the edit context (SEdit's main data structure), the charcode that was typed, and any extra arguments supplied in *FORM*. The extra arguments do not get evaluated, but are useful as keywords or flags, depending on how the command was invoked. The command function must return T if it handled the command. If the function returns NIL, SEdit will ignore the command and insert the character typed.

The optional arguments are used to add this command to SEdit's middle button menu. When the item is selected fromthe menu, the command function will be called as described above, with the charcode argument set to NIL.

KEY-NAME is a string to identify the key (combination) to be typed to invoke the command. For example "M-A" to represent the Meta-A key combination, and "M-C-A" for Meta-Control-A.

COMMAND-NAME is a string to identify the command function, and will appear in the menu next to the *KEY-NAME*.

HELP-STRING is a string to be printed in the prompt window when a mouse button is held down over the menu item.

After adding all the commands that you want, you must call Reset-Commands to install them.

For example:

```
(add-command "^U" (my-change-case t))

(add-command "^Y" (my-change-case nil))

(add-command "1,r" my-remove-nil

  "M-R" "Remove NIL"

  "Remove NIL from the selected structure"))

(reset-commands)
```

will add three commands. Suppose `my-change-case` takes the arguments *CONTEXT*, *CHARCODE*, and *UPPER-CASE*?. *UPPER-CASE*? will be set to T when `my-change-case` is called from Control-U, and NIL when called from Control-Y. `my-remove-nil` will be called with only *CONTEXT* and *CHARCODE* arguments when Meta-R is typed.

Below are some SEdit functions which are useful in writing new commands.

(Reset-Commands) [Function**]**

This function installs all commands added by **Add-Command**. SEdits which are open at the time of the **Reset-Commands** will not see the new commands; only new SEdits will have the new commands available.

(Default-Commands) [Function**]**

This function removes all commands added by **Add-Command**, leaving SEdit with its default set of commands. As in **Reset-Commands**, open SEdits will not be changed; only new SEdits will have the user commands removed.

(Get-Prompt-Window *CONTEXT***)** [Function**]**

This function returns the attached prompt window for a particular SEdit.

(Get-Selection *CONTEXT***)** [Function**]**

This function returns two values: the selected structure, and the type of selection, one of NIL, T, or :SUB-LIST. The selection type NIL means there is not a valid selection (in this case the structure is meaningless). T means the selection is one complete structure. :SUB-LIST means a series of elements in a list is selected, in which case the structure returned is a list of the elements selected.

(Replace-Selection *CONTEXT STRUCTURE SELECTION-TYPE* **)** [Function**]**

This function replaces the current selection with a new structure, or multiple structures, by deleting the selection and then inserting the new structure(s). The *SELECTION-TYPE* argument must be one of T or :SUB-LIST. If T the *STRUCTURE* is inserted as one complete structure. If :SUB-LIST, the *STRUCTURE* is treated as a list of elements, each of which is insertd.

16.1.13 Programmer's Interface

This programmer's interface to SEdit provides a way to call SEdit directly. This interface is sketchy and will change inthe future.

Getdef-Fn [Variable]

This function is called with the arguments *NAME*, *TYPE*, and *OLDDEF*, when SEdit needs to refetch the definition for the named object being edited. When SEdit is first started it gets passed the structure, so this function doesn't get called. But after completion, SEdit refetches because it doesn't know if the Edit Interface (File Manager) changed the definition upon installation. The function returns the new definition.

Fetch-Definition-Error-Break-Flag [Variable]

This flag, along with the error options listed below, determines what happens when the Getdef-Fn errors. The default value is NIL, causing errors to be suppressed. When set to T, the break will be allowed.

Getdef-Error-Fn [Variable]

This function is funcalled with the arguments *NAME*, *TYPE*, *OLDDEF*, and *PROMPT-WINDOW*, when the Getdef-Fn errors, independent of whether or not the break is suppressed. This function should return the structure to be used in place of the unavailable new definition.

Edit-Fn [Variable]

This function is funcalled with the selected structure as its argument from the Edit (M-O) command. It should start the editor as appropriate, or else generate an error if the selection is not editable.

Compile-Fn [Variable]

This function is funcalled with the arguments *NAME*, *TYPE*, and *BODY*, from the compile completion commands. It should compile the definition, *BODY*, and install the code as appropriate.

(SEdit *STRUCTURE PROPS OPTIONS***)** [Function]

This function provides a means of starting SEdit directly. *STRUCTURE* is the structure to be edited.

PROPS is a property list, which may specify the following properties:

:name - the name of the object being edited

:type - the file manager type of the object being edited. If NIL, SEdit will not call the file manager when it tries to refetch the definition it is editing. Instead, it will just continue to use the structure that it has.

:completion-fn - the function to be called when the edit session is completed. This function is called with the *CONTEXT*, *STRUCTURE*, and *CHANGED?* arguments. *CONTEXT* is SEdits main data structure. *STRUCTURE* is the structure being edited. *CHANGED?* specifies if any changes have been made, and is one of NIL, T, or :ABORT, where :ABORT means the user is aborting the edit and throwing away any changes made. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments that the function is applied to following the main arguments above.

:root-changed-fn - the function to be called when the entire structure being edited is replaced with a new structure. This function is called with the new structure as its argument. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments that the function is applied to following the structure argument.

OPTIONS is one or a list of any number of the followng keywords:

:fetch-definition-suppress-errors - If this option is provided, any error under the Getdef-Fn will be suppressed, regardless of the :fetch-definition-allow-errors option or the value of ***Fetch-Definition-Error-Break-Flag***.

:fetch-definition-allow-errors - If this option is provided, any error under the Getdef-Fn will be allowed to break.

:dontwait - This option specifies that the call to **SEdit** should return as soon as the editor is started, rather than waiting for a completion command.

:close-on-completion - This option specifies that SEdit cannot remain active for multiple completions. That is, the SEdit window cannot be shrunk, and the completion commands that normally leave the window open will in this case close the window and terminate the edit.

:compile-on-completion - This option specifies that SEdit should call the *Compile-Fn* to compile the definition being edited upon completion, regardless of the completion command used.

Fixed ARS

AR 7471 ---- You no longer have to Ctrl-X out of SEdit to compile. To compile, select either of two new SEdit commands. The C-C command (Ctrl -C) compiles and leaves the SEdit window open. The M-C-C command (Meta-Ctrl- C) compiles and closes the window.

AR 7783 ---- After typing Meta-O, you are prompted to "Select a type of dummy definition to install." The first option is "Optimizers". This title refers to XCL:DEFOPTIMIZERS, and has nothing to do with CL:OPTIMIZE.

AR 7786 ---- Square bracket characters '[' and ']' are not treated as special characters by SEdit, just as in Common Lisp.

[This page intentionally left blank]

Test script for SEdit version February 24, 1987

SEE CHANGE CONTROL FOR THE FILE SEDIT

FIXED:
7692 M-E breaks on numbers
7682 Ctrl-W with pending delete selection breaks
7705 M-H with extended selection breaks
7717 M-J breaks when nothing selected
7759 forgets package of mutator candidate
7731 M-J breaks with a sequence of numbers
7509 SEDIT.RESET needs IP fonts
7576 SEDIT.RELOAD shoud not be part of SEdit

Select a number and hit M-E, the result will be the number.

CTRL-W when there is a pending delete selection will do nothing. The easy test case is single-quote then ctrl-w.

Make an extended selection and hit M-H (help). Sedit will now say "Select function you want the arguments for"

Just after Starting SEdit, or any other time nothing is selected and/or there isno point, M-J used to break. Will now say "Select items to join."

SEdit now remembers the package of the mutate function candidate. Type in the atom abc. Get into the LISP package (M-P, LISP). Select the atom abc. Hit M-Z, and type IL:U-CASE, cr. Now hit M-Z again, the old SEdit would prompt with U-CASE, the new sedit will prompt with IL:U-CASE.

Select a sequence of numbers, or any sequence with a number first. Hit M-J. SEdit will say "Can't join numbers". SEdit will join AB 123 CD, though, as it used to.

With INTERPRESSFONTDIRECTORIES set to NIL, call SEDIT.RESET. It won't try to create IP fonts, and thus won't break.

The function SEDIT.RELOAD is not longer in the sysout.

,. wasn't recognized as a quote type. (SEDIT '(BQUOTE (A (\,. FOO) B] used to display as ‘(A (\,. FOO) B) and now will display as ‘(A ,.FOO B).

And now for all the shift selecting stuff:

Weren't able to Move select a structure into a pending delete selection. Now you can unless the structure overlaps the pending delete selection in some way. SEdit will say "Can't move a structure which overlaps the selection" if you try.

Can move select an object out of a quote, into any destination (exec, same sedit, different sedit), and the object will be replaced by a gap, which is selected pending delete.

Shift selecting something into a string didn't use to be completely undoable. Try Move selecting an atom into a string, then hit undo.

Move selecting something into a different destination when there is a main selection used to leave the selection messed up. Now it doesn't.

TWODINSPECTOR

By: Jan Pedersen (Pedersen.PA @ Xerox.ARPA)

The TWODINSPECTOR package provides a two-dimensional inspector window abstraction, very similar in form to the standard one-dimensional inspector but laid out in rows and columns, instead of just rows.

The top level function is TWODINSPECTW.CREATE

(TWODINSPECTW.CREATE *DATUM ROWPROPS COLUMNPROPS FETCHFN STOREFN VALUECOMMANDFN ROWPROPCOMMANDFN COLUMNPROPCOMMANDFN TITLE TITLECOMMANDFN WHERE TOPRIGHT*) [Function]

Datum is the object to be inspected. Rowprops is a list of properties of the datum which will be laid out vertically, or a function which will be called with datum as an argument and returns such a list. Similarly, columnprops is a list of properties of the datum which will be laid out horizontally, or a function which will be called with datum as an argument and returns such a list. Each pair (rowprop, columprop) specifies a cell of the twodimensional inspector window. Fetchfn is a function which if called with arguments datum, rowprop, and columprop returns the value in that cell. Storefn is a function which if called with arguments newvalue, datum, rowprop, and columprop stores newvalue in the cell.

The cells of the inspector window are selectable. If valuecommandfn is given, it must be a function which will be called with arguments cellvalue, rowprop, columnprop, datum, and twodinspectwindow when the cell specified by (rowprop, columnprop) is selected. A default valuecommandfn is provided which allows the cellvalue to be inspected, set, or bound to the litatom IT.

Similarly the rowprops and the columnprops themselves are selectable. If rowpropcommandfn is given it must be a function which will be called with args rowprop, datum, and twodinspectwindow when rowprop is selected. If columnpropcommandfn is given it must be a function which will be called with args columnprop, datum, and twodinspectwindow when columnprop is selected. No default rowpropcommandfn or columnpropcommandfn is provided. If rowpropcommandfn is not given, the rowprops will not be selectable. Similarly, If columnpropcommandfn is not given, the columnprops will not be selectable.

Title will be the title for the window -- a default is provided. Titlecommandfn is a function which will be called with the single argument twodinspectwindow if the middle button is depressed in the title bar of the window.

Where may be a window, in which case it will be used as at least part of the twodinspector (the twodinspector is composed of five window), This is especially useful if where is the result of a previous call to TWODINSPECTW.CREATE. The dimensions of where will not be used to position the twodinspector unless topright is NIL.

Where may also be region or a position specifying the lower left hand corner of the twodinspector. If where is NIL, the user will be prompted for a position.

Topright allows the user to specify the top right-hand corner of the twodinspector. Topright must be a position, and if given overrides any specification which may have been provided by the argument where.

Returns the main window of an attached window group.

The arguments to TWODINSPECTW.CREATE are cached on windowprops of the same name on the returned main window.

Several functions are provided for use in the various command functions.

(TWODINSPECT.REDISPLAY *TWODINSPECTW SOMEROWPROPS*ˆ
SOMECOLUMNPROPS) [Function]

Redisplay selected cells of twodinspectw. Somerowprops may either be a single rowprop, a list of rowprops, or NIL. Somecolumnprops may either be a single columnprop, a list of columnprops, or NIL. If either are NIL the entire twodinspectw is recomputed and redisplayed. Otherwise, the cells specified by the cross product of somerowprops and somecolumnprops are redisplayed, possibly forcing the entire twodinspectw to redisplay if the printed representation of a cell overflows its column width.

(TWODINSPECT.REPLACE *TWODINSPECTW ROWPROP COLUMNPROP*ˆ
NEWVALUE) [Function]

Replaces the cell specified by (rowprop, columprop) with newvalue and updates the display.

(TWODINSPECT.SELECTITEM *TWODINSPECTW ROWPROP COLUMNPROP*] [Function]

Selects the cell specified by (rowprop, columprop). That cell is inverted and put on the window prop SELECTION of twodinspectw. If either of rowprop or columprop is NIL, then the current selection is simply deselected.

(TWODINSPECT.SELECTROWPROP *TWODINSPECTW ROWPROP*] [Function]

Selects rowprop. If rowprop is NIL, then the currently selected rowprop is deselected.

(TWODINSPECT.SELECTCOLUMNPROP *TWODINSPECTW* COLUMNPROP*]* [Function]

Selects columprop. If rowprop is NIL, then the currently selected columprop is deselected.

Note: there is no provision for redisplaying selected row or column props -- although this may be effected by redisplaying the entire twodinspectw.

Since the Twodinspector windows differ stylistically from the standard inspector windows, a stylistically similar onedinspector window is also provided.

(ONEDINSPECTW.CREATE *DATUM PROPS FETCHFN STOREFN VALUECOMMANDFN^ PROPCOMMANDFN TITLE TITLECOMMANDFN WHERE TOPRIGHT*) [Function]

Datum is the object to be inspected.Props is a list of properties of the datum which will be laid out horizontally, or a function which will be called with datum as an argument and returns such a list. Each prop specifies a cell of the onedimensional inspector window. Fetchfn is a function which if called with arguments datum, and prop returns the value in that cell. Storefn is a function which if called with arguments newvalue, datum, and prop stores newvalue in the cell.

The cells of the inspector window are selectable. If valuecommandfn is given, it must be a function which will be called with arguments cellvalue, prop, datum, and onedinspectwindow when the cell specified by prop is selected. A default valuecommandfn is provided which allows the cellvalue to be inspected, set, or bound to the litatom IT.

Similarly the props themselves are selectable. If propcommandfn is given it must be a function which will be called with args prop, datum, and onedinspectwindow when prop is selected. No default propcommandfn is provided. If propcommandfn is not given, the props will not be selectable.

Title will be the title for the window -- a default is provided. Titlecommandfn is a function which will be called with the single argument onedinspectwindow if the middle button is depressed in the title bar of the window.

Where may be a window, in which case it will be used as at least part of the onedinspector (the onedinspector is composed of three window), This is especially useful if where is the result of a previous call to ONEDINSPECTW.CREATE or TWODINSPECTW.CREATE. The dimensions of where will not be used to position the onedinspector unless topright is NIL.

Where may also be region or a position specifying the lower left hand corner of the onedinspector. If where is NIL, the user will be prompted for a position.

Topright allows the user to specify the top right-hand corner of the onedinspector. Topright must be a position, and if given overrides any specification which may have been provided by the argument where.

Returns the main window of an attached window group.

The arguments to ONEDINSPECTW.CREATE are cached on windowprops of the same name on the returned main window.

(ONEDINSPECT.REDISPLAY *ONEDINSPECTW SOMEPROPS*) [Function]

Redisplay selected cells of onedinspectw. Someprops may either be a single prop, a list of props, or NIL, in which case the entire onedinspectw is recomputed and redisplayed. Otherwise, the cell(s) specified by the someprops are redisplayed, possibly forcing the entire someprops to redisplay if the printed representation of a cell overflows the column width.

(ONEDINSPECT.REPLACE *ONEDINSPECTW PROP NEWVALUE*) [Function]

Replaces the cell specified by prop with newvalue and updates the display.

(ONEDINSPECT.SELECTITEM *ONEDINSPECTW PROP*) [Function]

Selects the cell specified by prop. That cell is inverted and put on the window prop SELECTION of onedinspectw. If prop is NIL, then the current selection is simply deselected.

(ONEDINSPECT.SELECPROP *ONEDINSPECTW PROP*) [Function]

Selects prop. If prop is NIL, then the currently selected prop is deselected.

```
EASTASIA:
      The CDROM came with CJK cross reference mappings for standards such as
KSC5601,
      GB2312, JIS0208, etc. to Unicode 2.0.
      However, these particular mappings are now obsolete and have been
removed as per
      this note from Unicode.org:
         The entire former contents of this directory are obsolete and have
been
       moved to the OBSOLETE directory.  The latest information may be found
       in the Unihan data files in the latest Unicode Character Database.
       August 1, 2001.
    The current set of mappings are available from
           https://unicode.org/Public/UNIDATA/Unihan.zip
    The format of these files is given in https://unicode.org/reports/tr38/
```

This Unicode directory contains mapping files extracted from the CDROM that came with the Unicode 3.0 book (2000).

The Xerox subdirectory contains mappings from the Xerox character encoding (version XC1-3-3-0, 1887) into Unicode 3.0.   standard into Unicode. That is the version of XCCS corresponding to the fonts in the Medley system. The Xerox mappings did not come from the Unicode CDROM, they were constructed by combining and constrasting information from a binary file (xerox>XCCStoUni) of unknown provenance with code mappings scraped from the Wikipedia page https://en.wikipedia.org/wiki/Xerox_Character_Code_Standard in July 2020. Both sources were errorful and incomplete, so many of the mappings were hand corrected. There are still missing mappings, and there still may be errors.

EASTASIA:
      The CDROM came with CJK cross reference mappings for standards such as KSC5601,
      GB2312, JIS0208, etc. to Unicode 2.0.
      However, these particular mappings are now obsolete and have been removed as per
      this note from Unicode.org:
         The entire former contents of this directory are obsolete and have been
       moved to the OBSOLETE directory. The latest information may be found
       in the Unihan data files in the latest Unicode Character Database.
       August 1, 2001.
    The current set of mappings are available from
          https://unicode.org/Public/UNIDATA/Unihan.zip
    The format of these files is given in https://unicode.org/reports/tr38/

ISO8859:
      These are the mapping tables of the ISO 8859 series (1 through 16)

VENDORS:
      Miscellaneous mapping tables for small codesets, typically provided
      by vendors.

TCVN:
      Chu Nom mapping & database.

Always consult www.unicode.org for updates and changes to these files.