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 not 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. This is filled in by the system.

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. This is filled in by the system.

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

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

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.

The Call-C-Function MISCN opcode

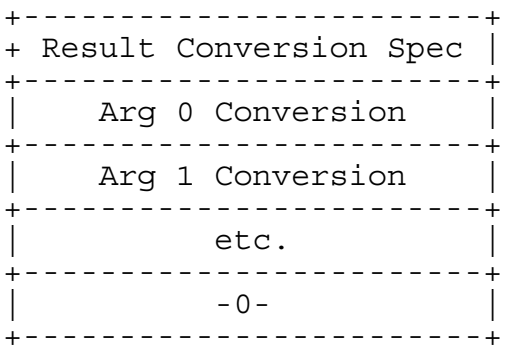
This opcode calls the specified C function, performaing conversion of arguments and result as needed, and returning an indication of any errors it encounters.

(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:



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 signall*ing,* 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 sub-class 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 re-constituted 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)
.....
(CDDDR x)

(caar 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	most-negative-fixnum
MAX.SMALLP	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)

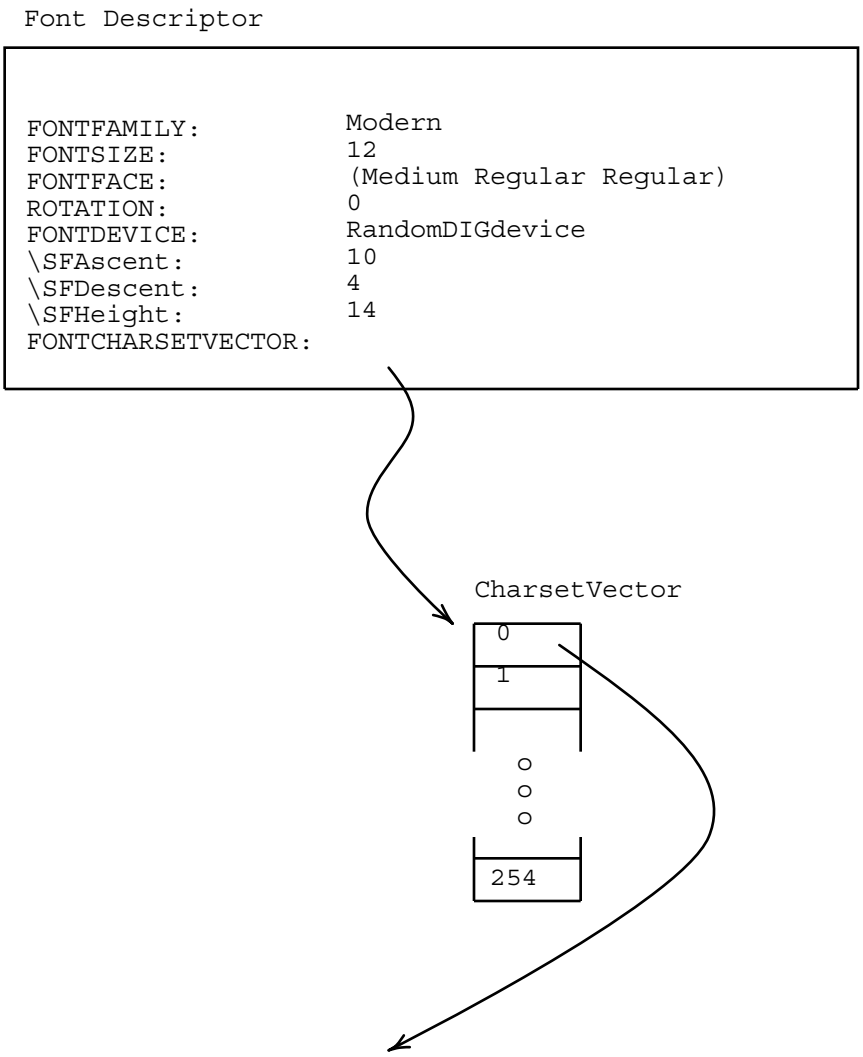
Note: In Intermezzo (and in some other operating systems), characters with the eighth bit set were considered "meta" characters. In the Koto release, however, "meta" means character set 1, and the meta key produces characters with the 400Q bit set, not 200Q.

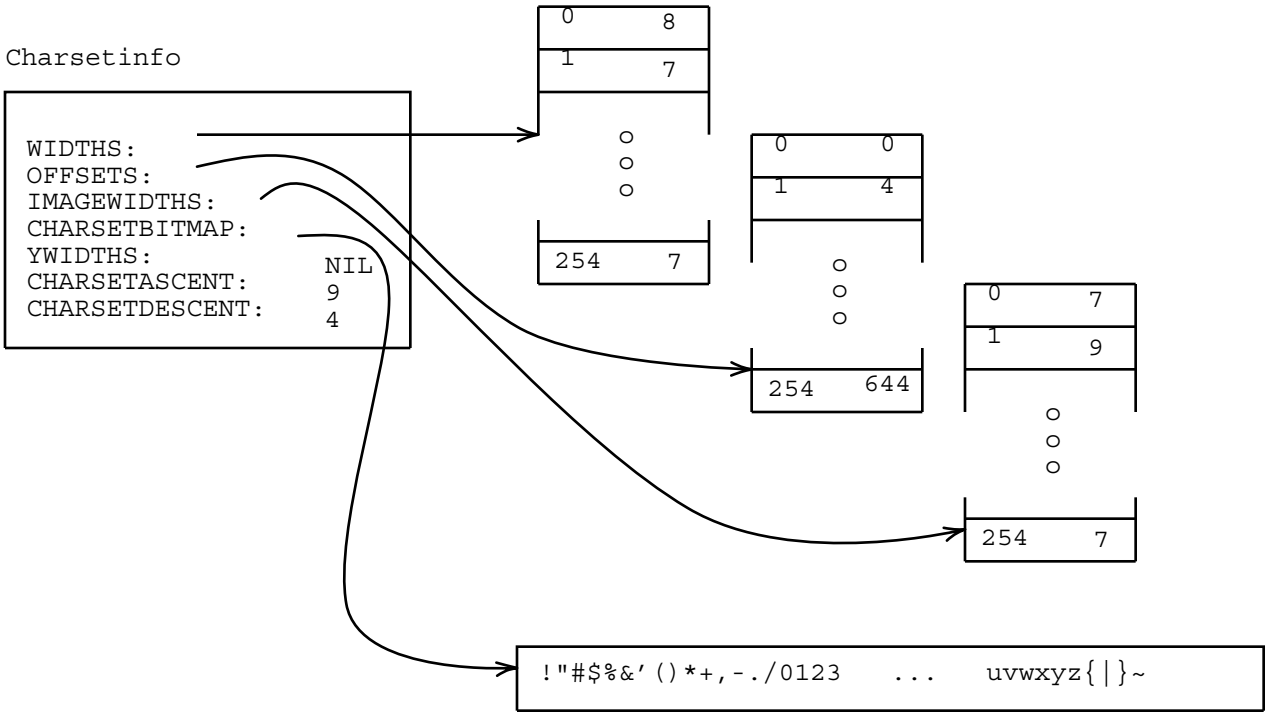
Internals

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.

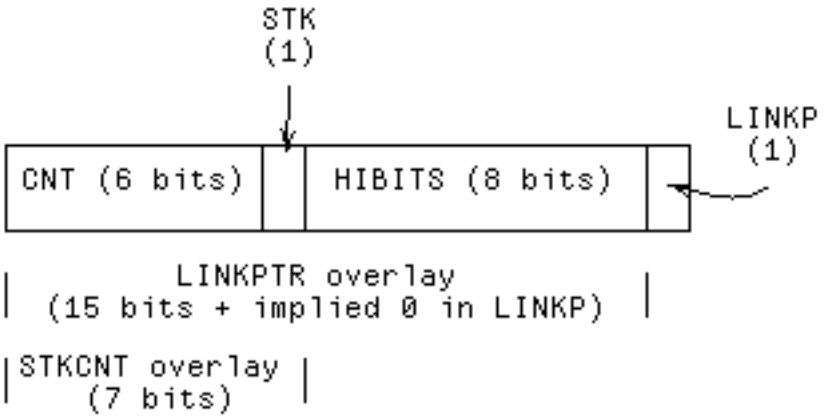
\HTBIGCOUNT is a linearly searched table of big ref count entries; new pages are allocated as needed.

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...

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 \GETBASEs 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 BLOCKRECORDs and DATATYPEs.

(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 [Function]
 RUNLENGTH NORAIDFLG)

"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 [Function]
RUNLENGTH NORAIDFLG)

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.

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:

Error in IMAGEOBJ 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


/************************************************************************/
/*
*/
/*    Copyright 1989, 1990 Venue                                        *
/
/*
*/
/*    This file was work-product resulting from the Xerox/Venue         */
/*    Agreement dated 18-August-1989 for support of Medley.             */
/*
*/
/************************************************************************/
```

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		

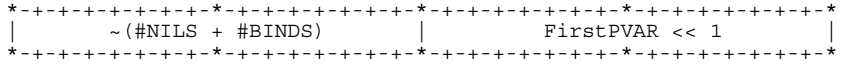
push binding mark, bind variables, popping values of stack. [required]

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:



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 :
```

#	name
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 ”Λ

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 exatly 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 `LineStart`s are the exception. `LineStart`s 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 `LineStart`s 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.).

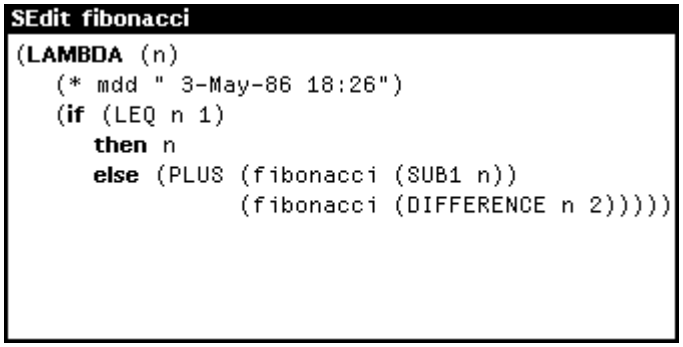
A Sample Session

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

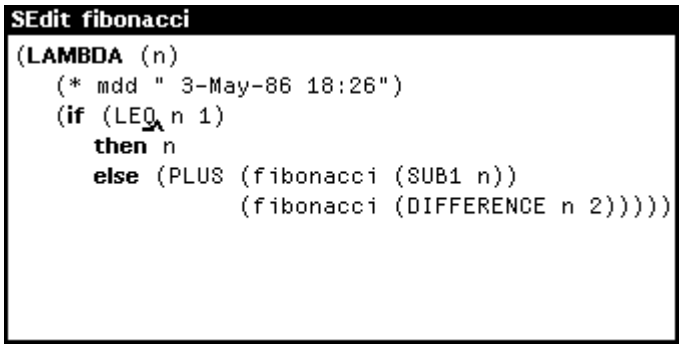
chosen insertion point; `Delete`, which deletes the selected material; and `Move`, which copies the selected material and then deletes it

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:

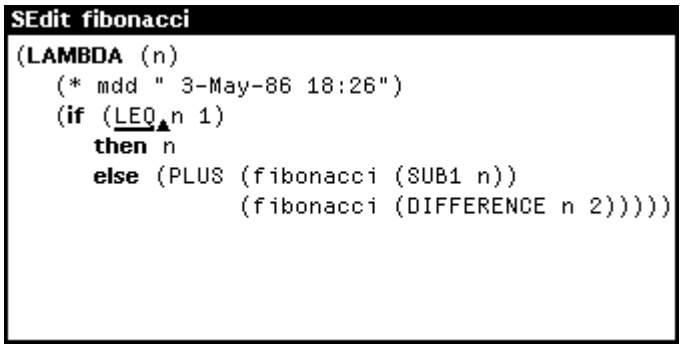


Since SEdit knows that this is Interlisp code, it uses some formatting rules specific to Lisp as well as its general rules for formatting lists, atoms, and strings. Formatting rules specify visual presentations as the positioning of text strings (in any font) and arbitrary bitmaps.

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:



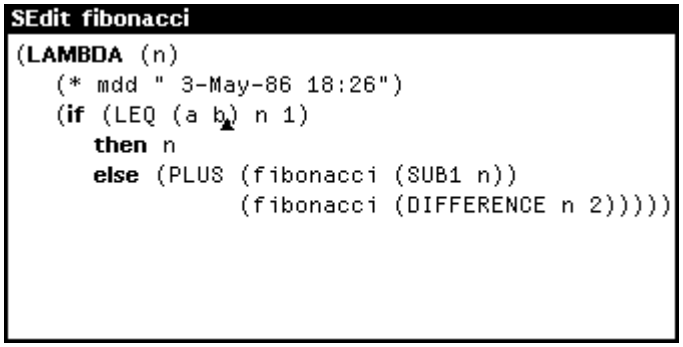
and any characters now typed would be appended to the atom `LEQ`. On the other hand, middle-clicking in the same place produces:



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

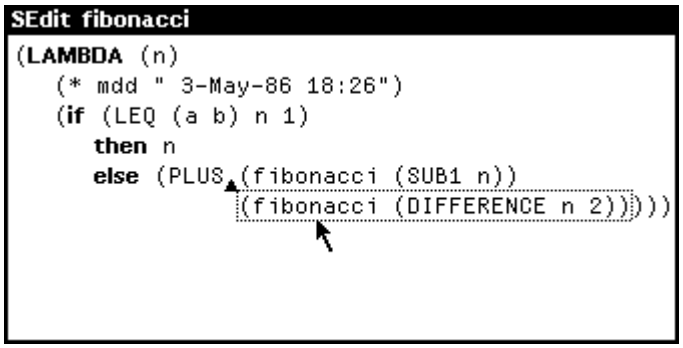
a) middle-click after `PLUS` (sets the insertion point)

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).

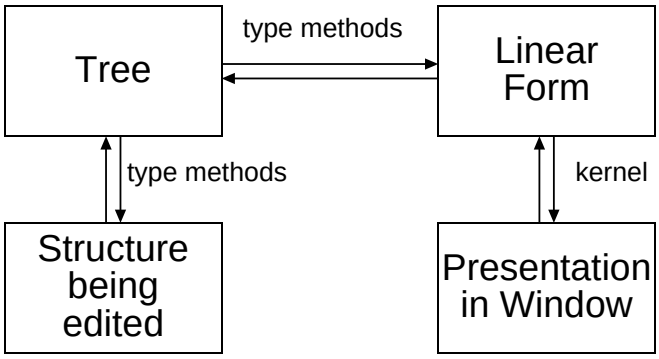
SEdit is installed by calling the function `(EDITMODE 'SEDIT)`, whereupon it becomes the system display editor and will be used whenever the user asks to display edit a structure (through edit commands, Masterscope searches, inspectors, browsers, or whatever). The context remains active until the edit window is closed (although the keyboard process disappears while the window is shrunk).

An Overview of the Data Structures

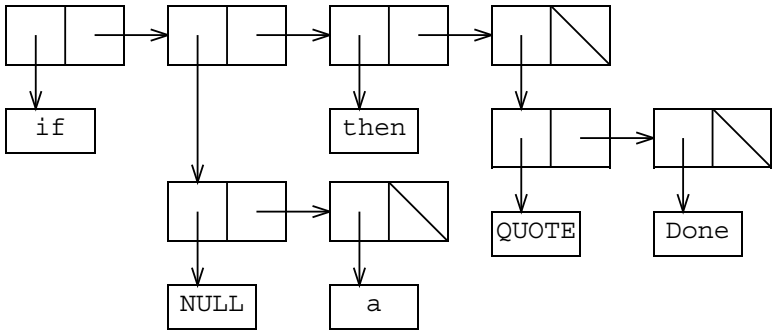
The SEdit user is aware of two data structures: the structure being edited, and the window in which it is presented. Internally, SEdit maintains two additional data structures: the *tree*, which is a representation of the structure being edited, and the *linear form*, which is a representation of the 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.

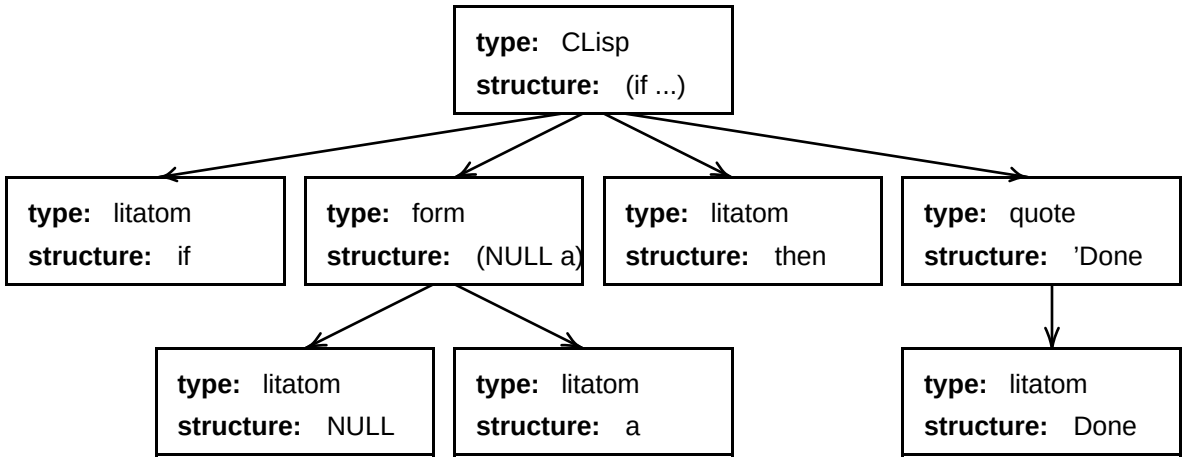
These four data structures are all representations of the same data. The constraints between them are maintained as follows:



As an example, consider editing the list structure `(if (NULL a) then (QUOTE Done))`. The underlying structure is composed of cons cells and atoms:



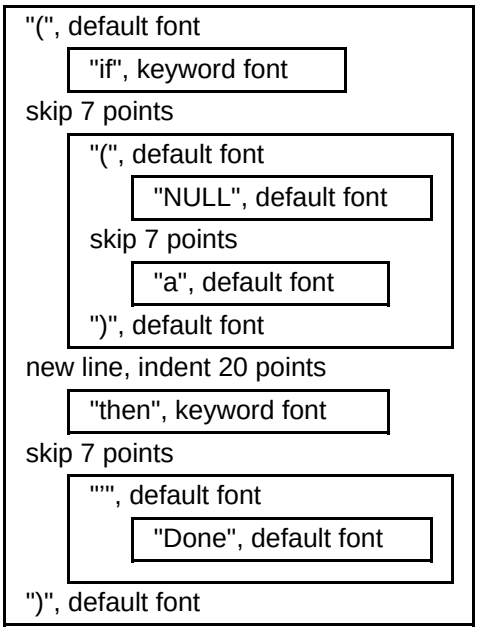
SEdit's tree will look like this:



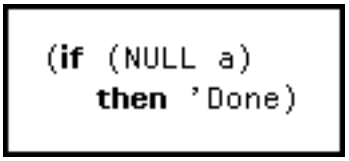
Note that each node in the tree has a type, indicating the type of the structure, and a field containing the structure itself. Note also that the types need not correspond directly with Interlisp's type

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:



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:



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.

SEdit Internal Documentation

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:

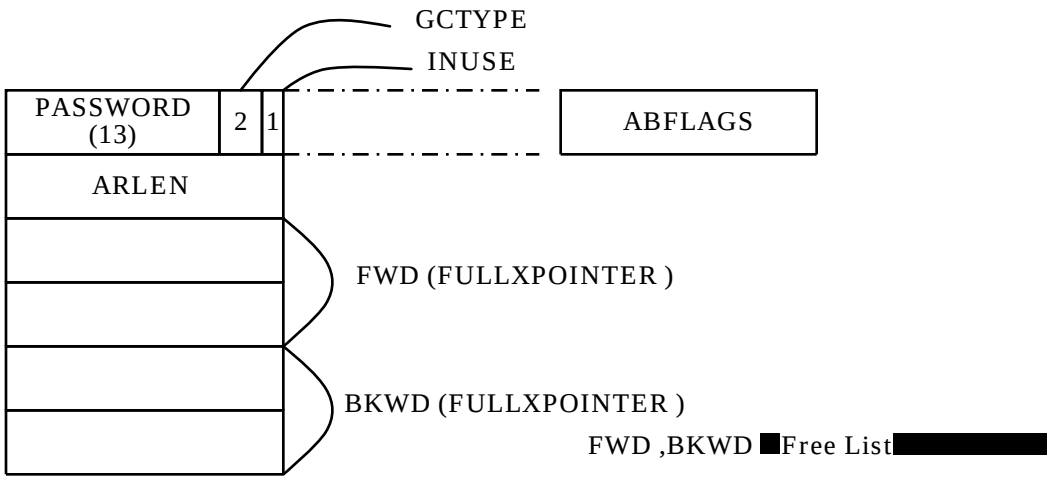
> [inconsistency: note-change actually maintains the queue in order of decreasing depth (despite its comments to the contrary), and relinearize-where-necessary reverses it before it looks at it. this turns out to be fine, since it wants the queue in decreasing depth for the next step. ought to fix the comments though...]

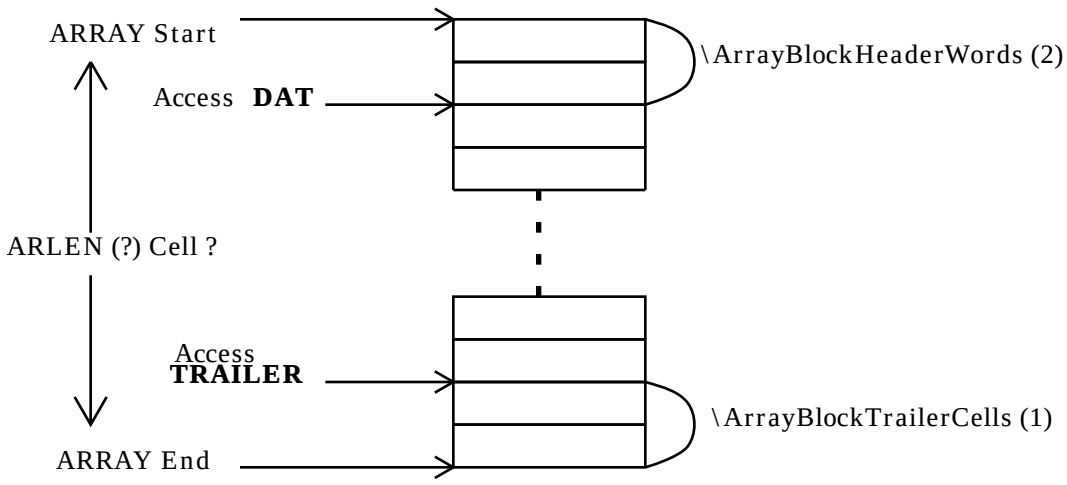
1. for each node on the queue (from top to bottom), assign-format to its super node and add it to the queue (unless it's already in the queue) and then assign-format to it.
2. for each node now in the queue (from bottom to top) recompute its width estimates, and if they've changed push its super on the 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.

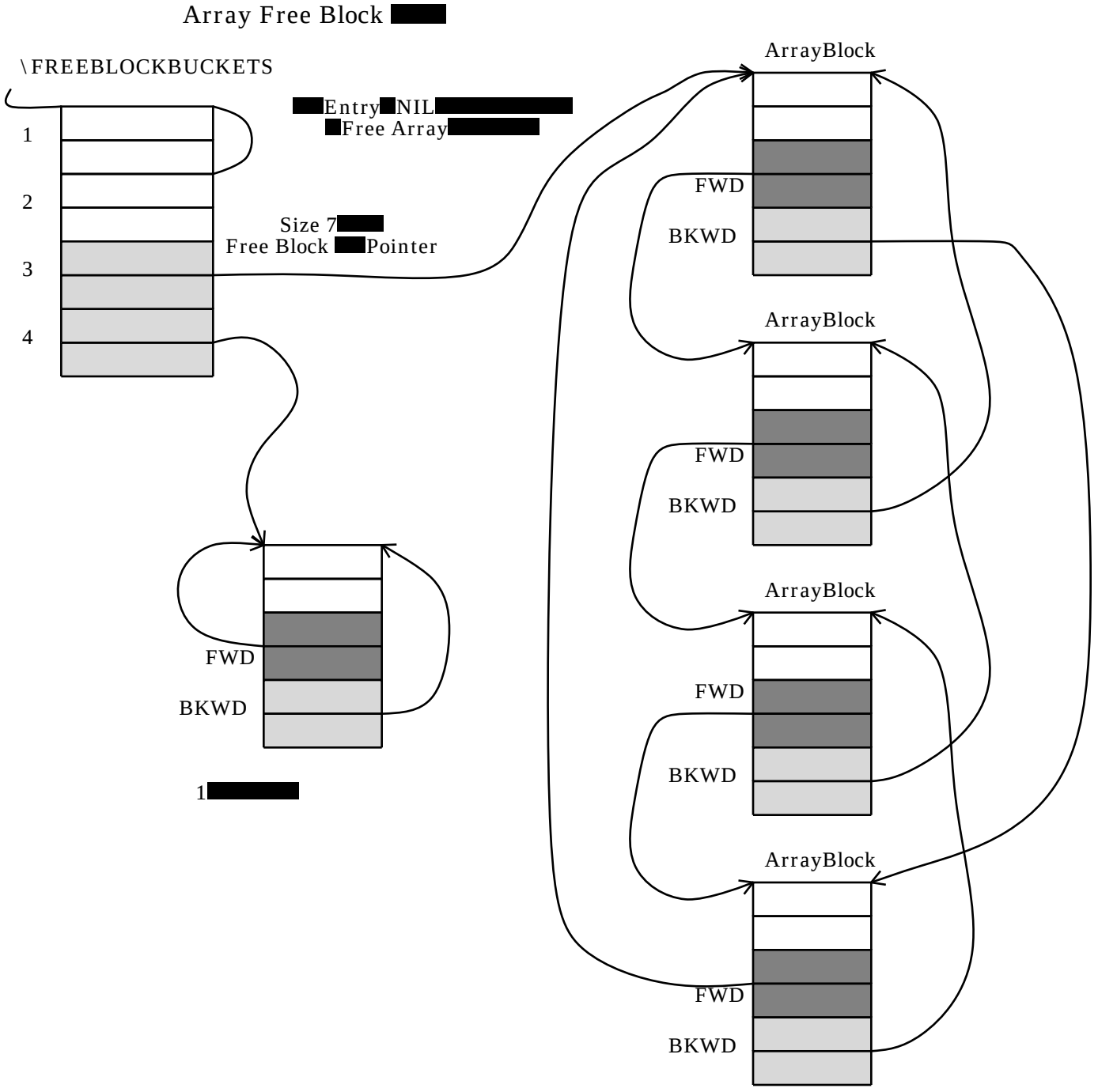
relinearizing a node is guaranteed to call its linearize method. In addition, it will call the linearize method of any subnode which (a) has changed, or (b) has been moved (relative to the right margin) so

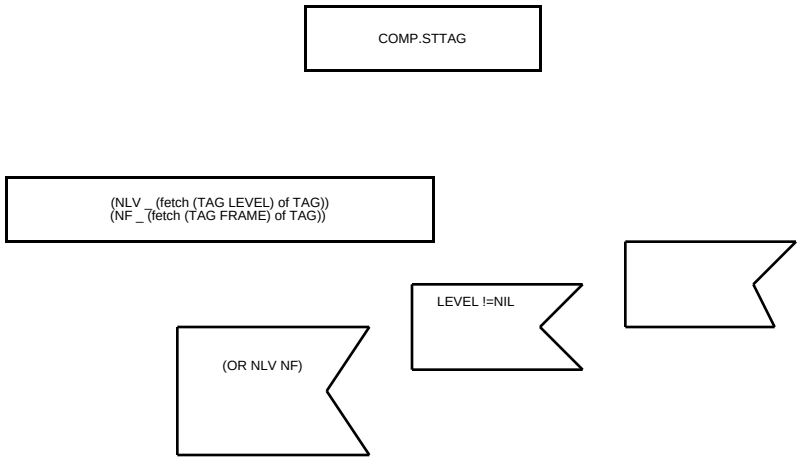
that it's linear form is no longer likely to be appropriate (the test for this is at the beginning of generate-linear-form) (and so on recursively into its subnodes). (and as i mentioned before, there's a whole extra story about how changes to the linear form are used to determine changes to the screen; 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





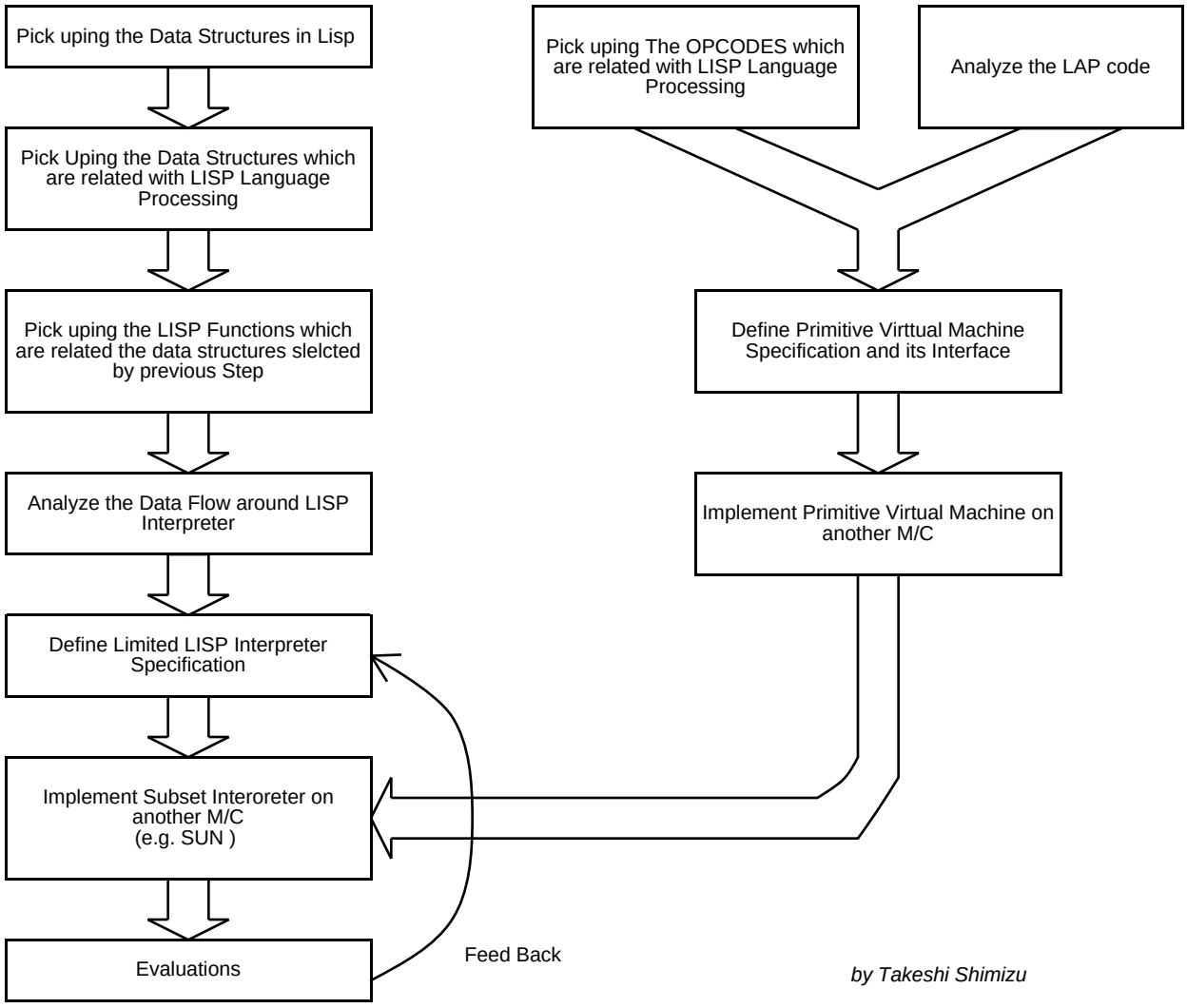

Array Free Block
\FREEBLOCKBUCKETS
1
2
3
4
Size 7
Free Block Pointer
ArrayBlock
FWD
BKWD
ArrayBlock
FWD
BKWD
ArrayBlock
FWD
BKWD
ArrayBlock
FWD
BKWD
FWD
BKWD


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		



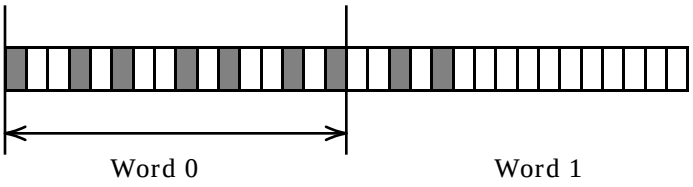
Interlisp-D Machine Development Concept (Primary Step)

[redacted]

Interlisp[redacted]BITMAP[redacted]

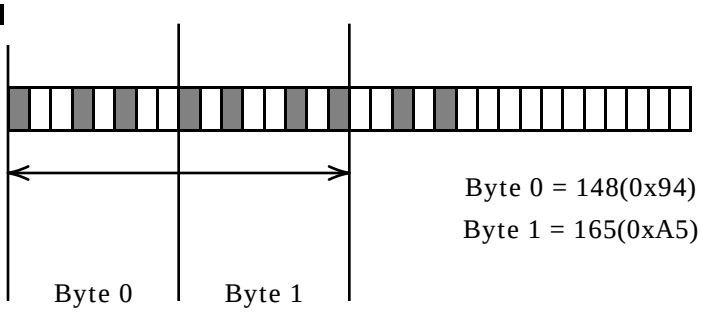
BitMap[redacted] = 0 = 1

[redacted]

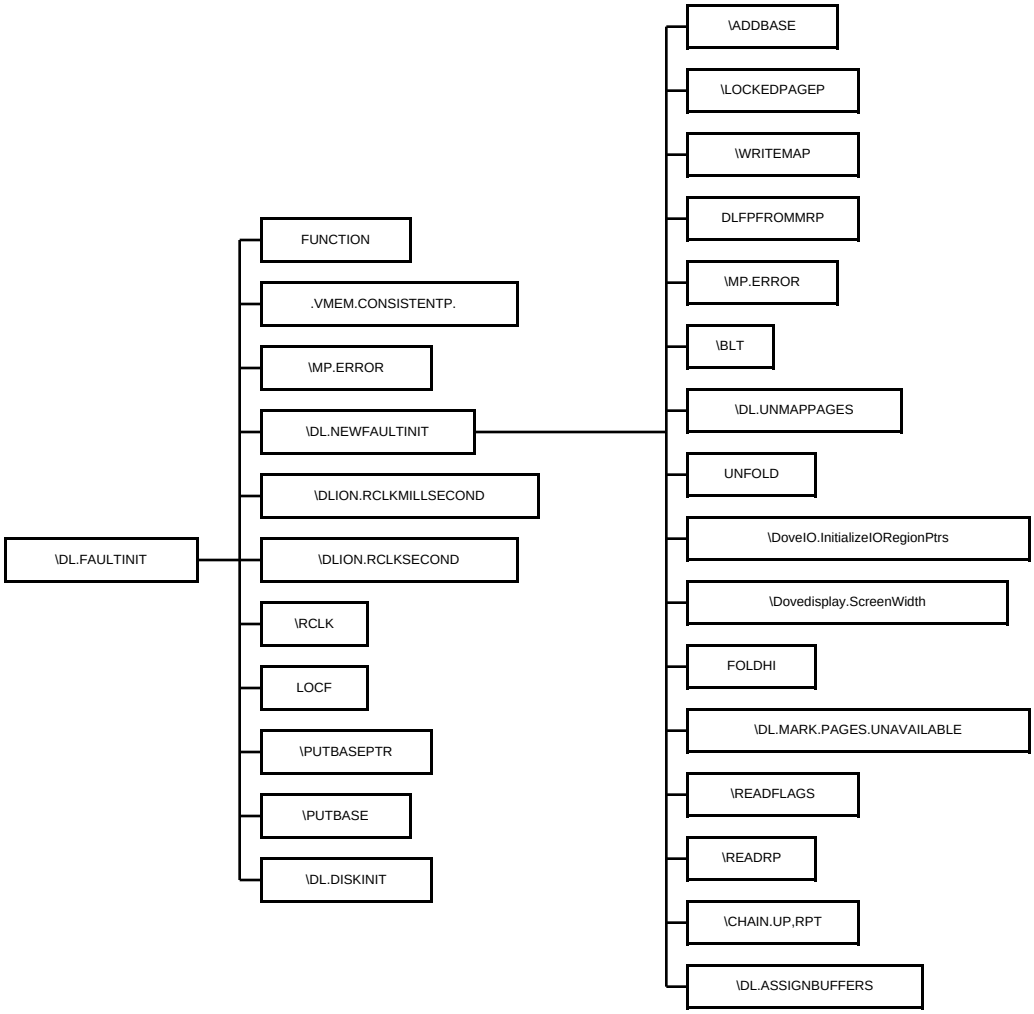


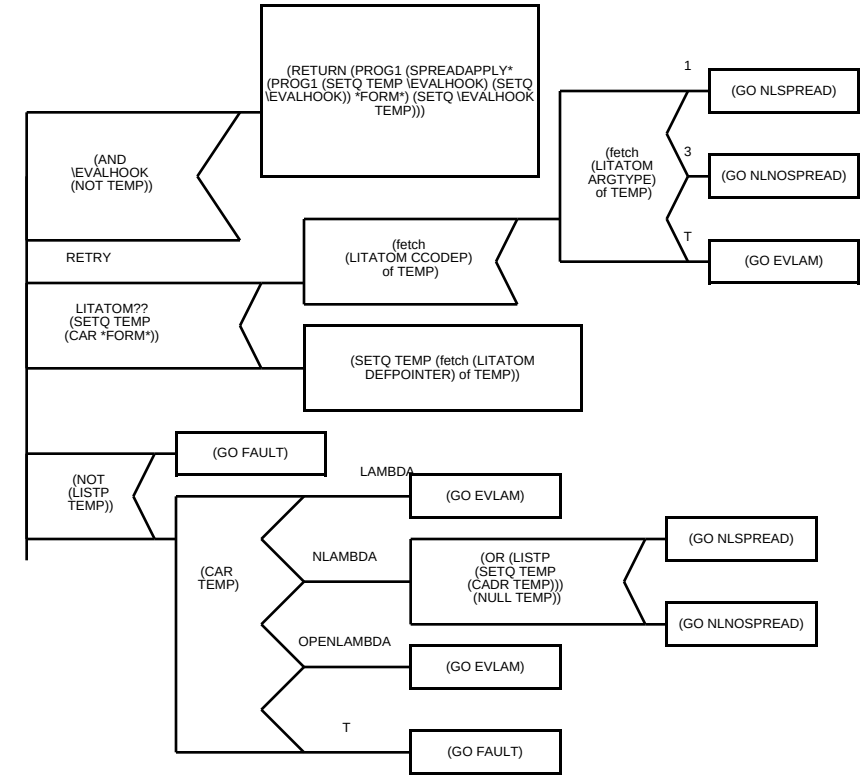
Word Access = 38053(0x94A5)

[redacted]



MSB LSB


\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



EVLAM

((*ARGVAL* 0) (*TAIL* *FORM*) (*FN* (CAR *FORM*)))

(RETURN (.EVALFORM.))

NLSPREAD

(RETURN (SPREADAPPLY (CAR *FORM*) (CDR *FORM*)))

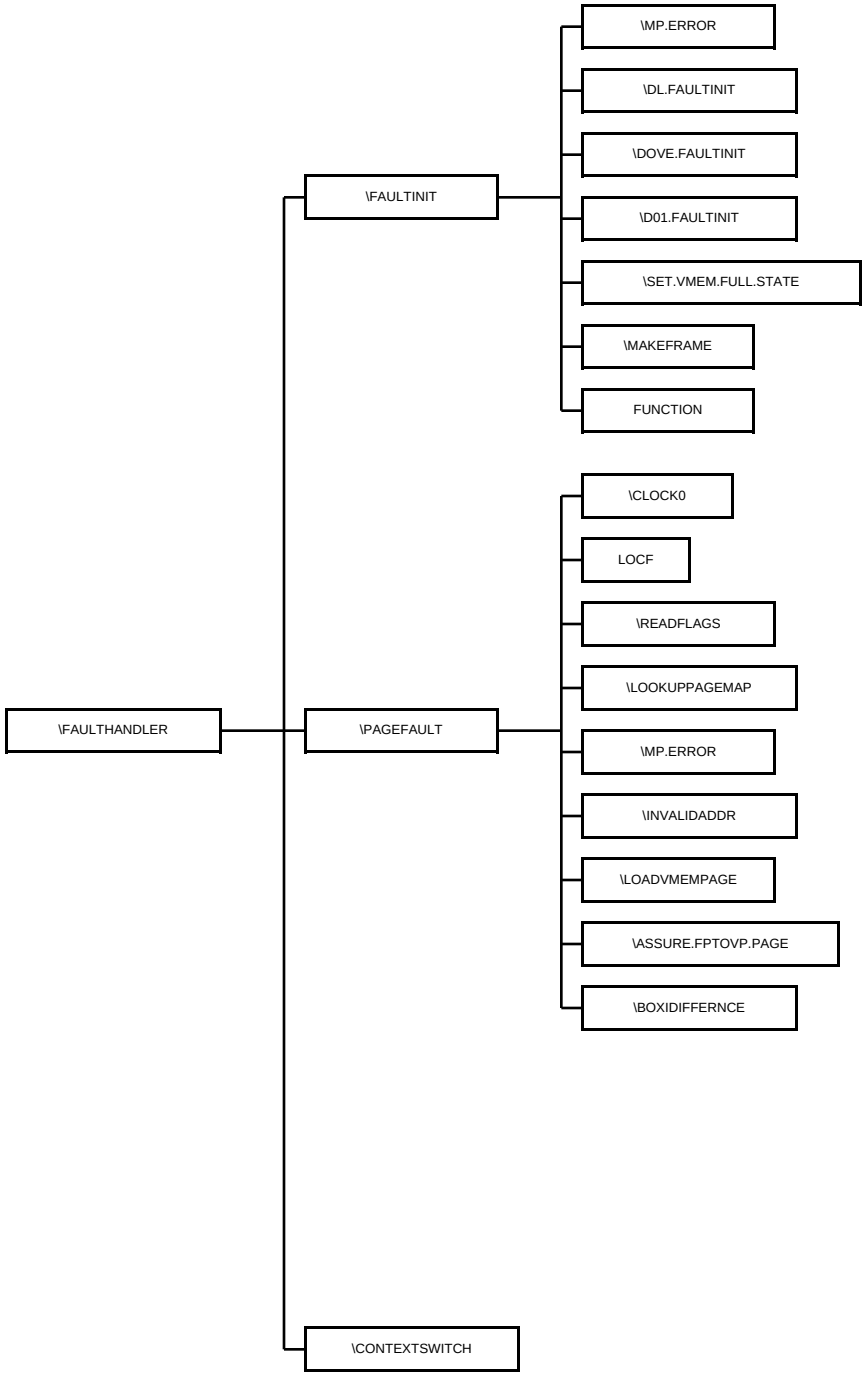
NLNOSPREAD

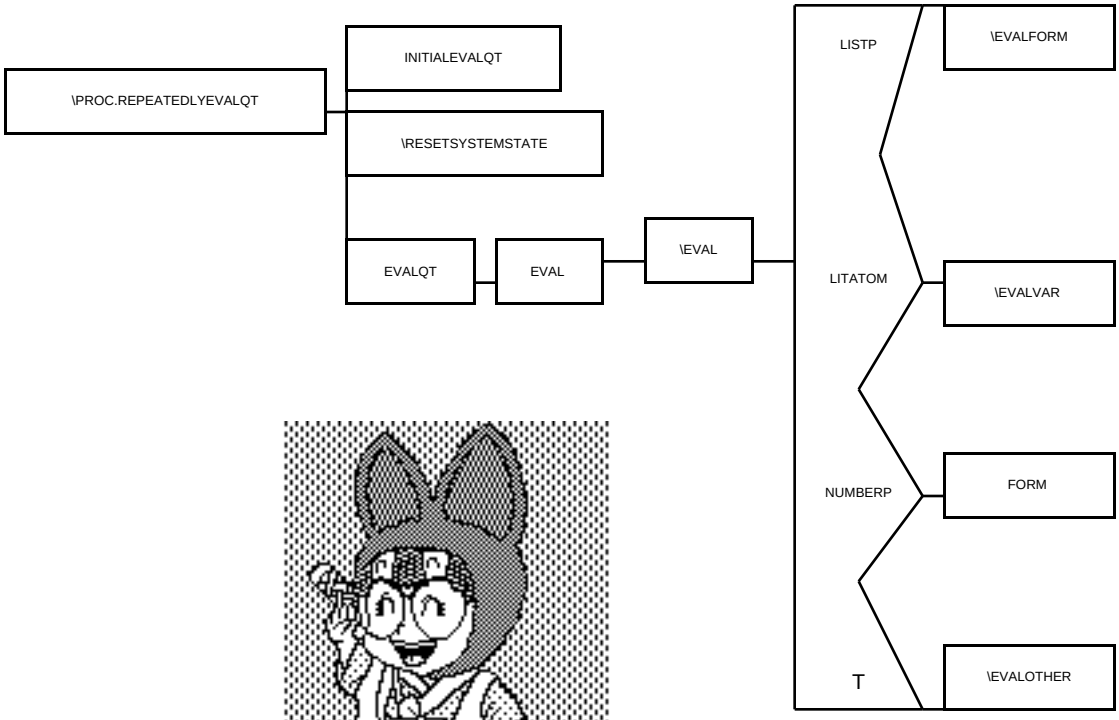
(RETURN (SPREADAPPLY* (CAR *FORM*) (CDR *FORM*)))

FAULT

CLISP PROCESSING

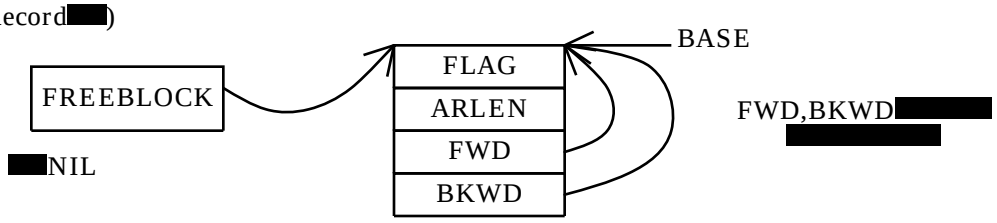
(GO RETRY)


\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

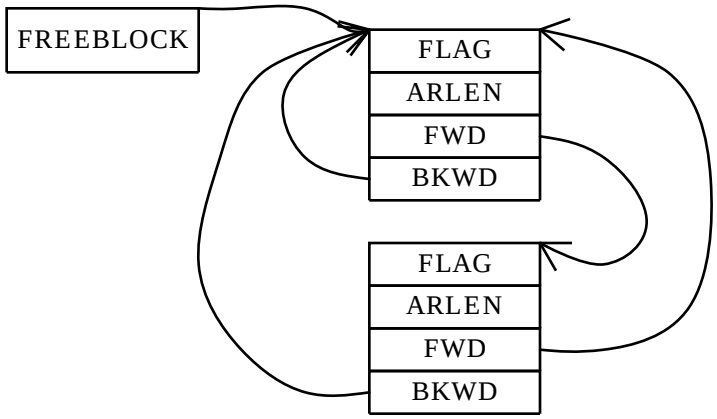

\PROC.REPEATEDLYEVALQT
INITIALEVALQT
\RESETSYSTEMSTATE
EVALQT
EVAL
\EVAL
LISTP
\EVALFORM
LITATOM
\EVALVAR
NUMBERP
FORM
T
\EVALOTHER

\LINKBLOCK

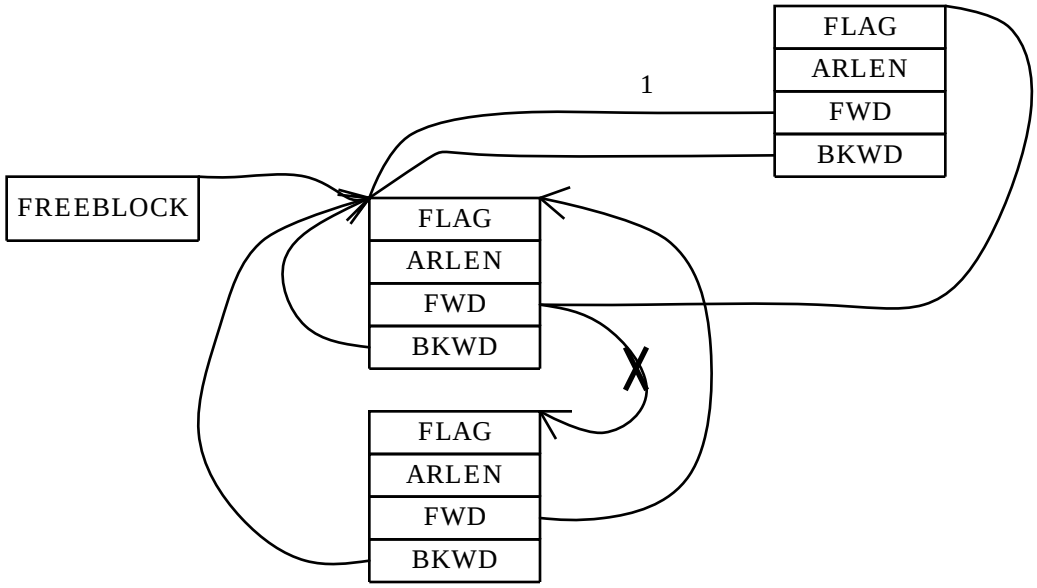
Case 1 ([redacted]Record[redacted])

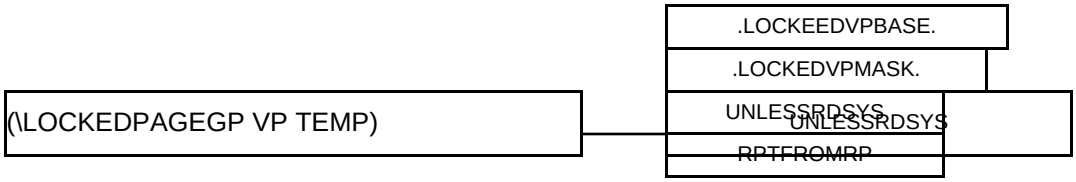


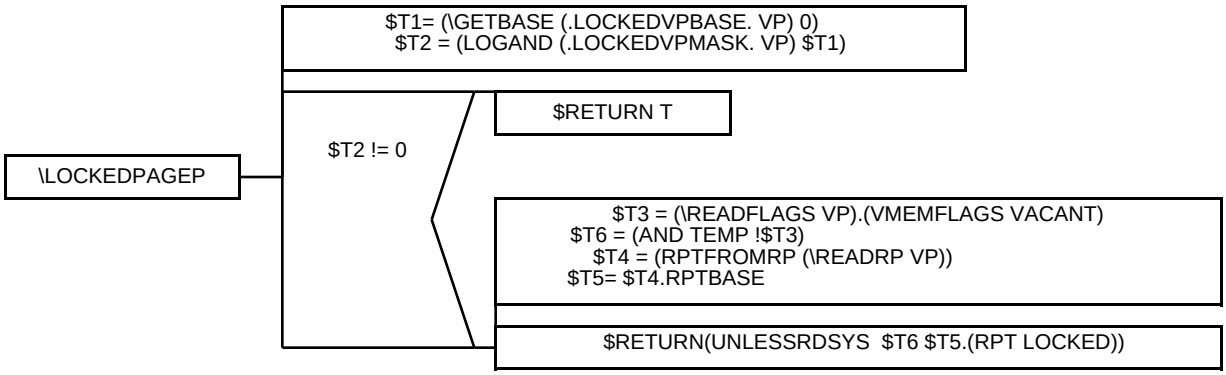
Case 2 LINK[redacted]



X

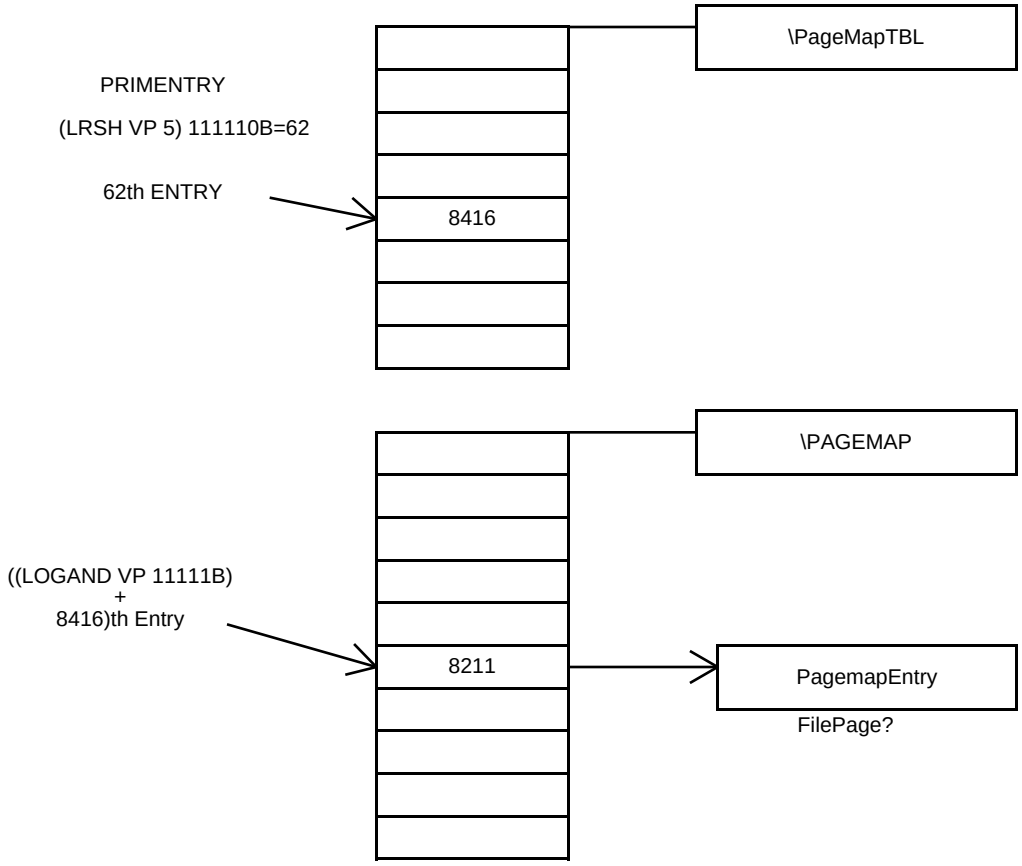



(\LOCKEDPAGEGP VP TEMP)
.LOCKEEDVPBASE.
.LOCKEDVPMASK.
UNLESSRDSYS
UNLESSRDSYS
RPTFROMRP

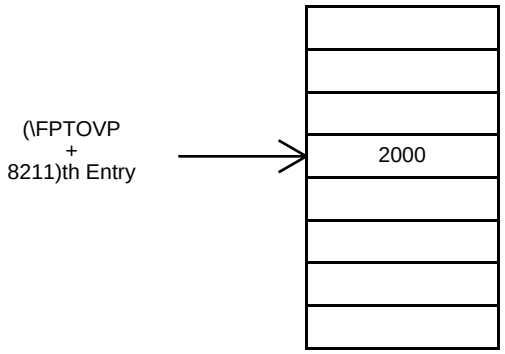

$T1= (\GETBASE (.LOCKEDVPBASE. VP) 0)
$T2 = (LOGAND (.LOCKEDVPMASK. VP) $T1)
\LOCKEDPAGEP
$T2 != 0
$RETURN T
$T3 = (\READFLAGS VP).(VMEMFLAGS VACANT)
$T6 = (AND TEMP !$T3)
$T4 = (RPTFROMRP (\READRP VP))
$T5= $T4.RPTBASE
$RETURN(UNLESSRDSYS $T6 $T5.(RPT LOCKED))

\LOOKUPPAGEMAP

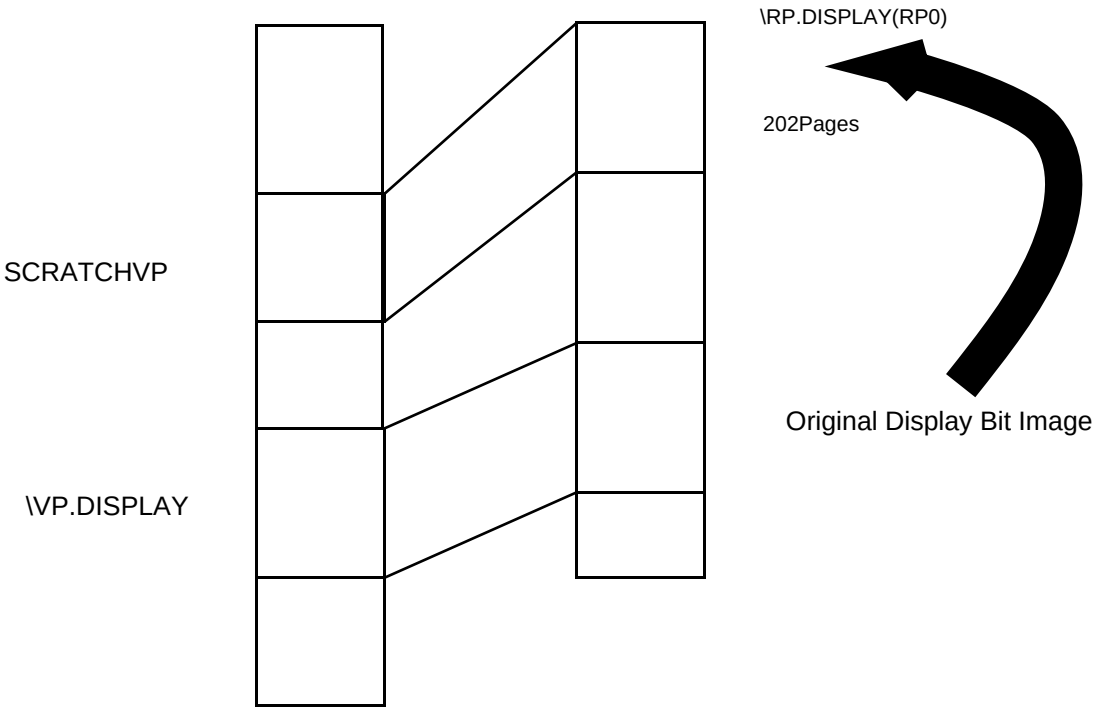
VP= 11111010000B=2000



Operations by Using \FPTOVP

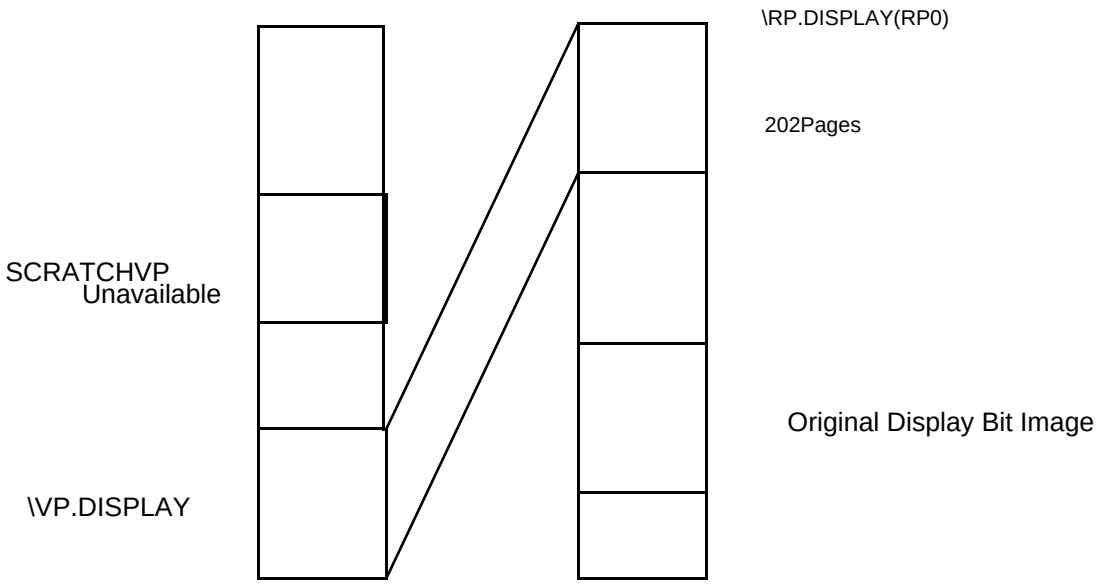


MOVE DIsplay Region First

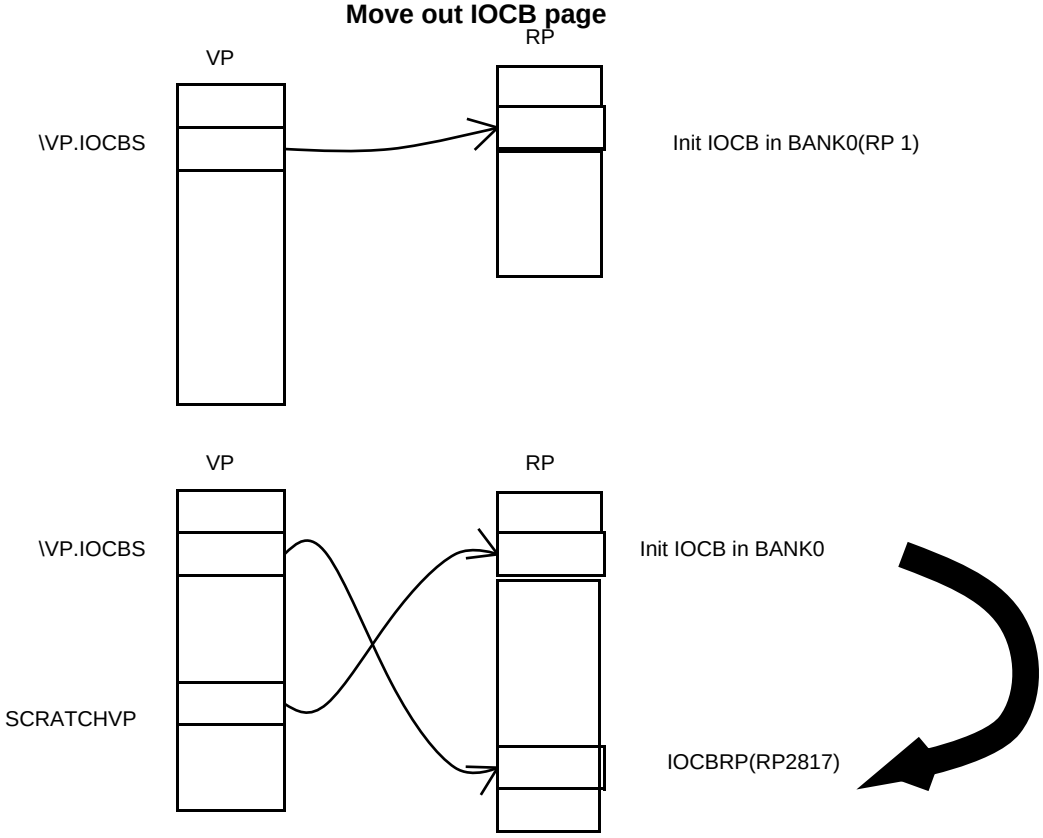


(\BLT SCRATCHBASE (create POINTER #PAGE _\VP.DISPLAY) NWORD)

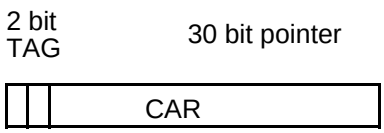
MOVE DIsplay Region Second



Move out IOCB page



(\BLT \IOCBPAGE SCRATCHBASE WORDSPERPAGE)

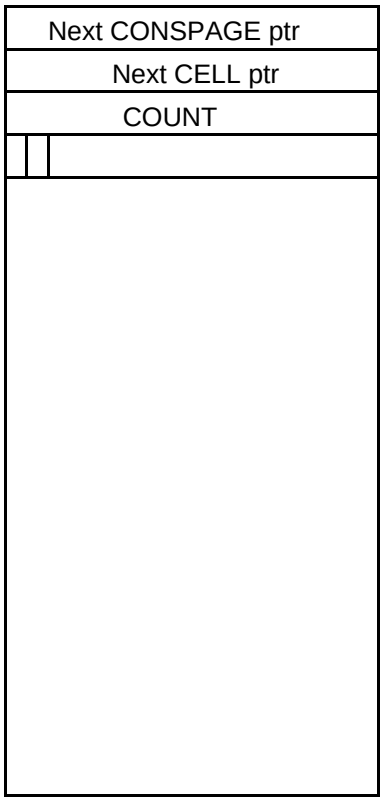


00 : CDR is NIL

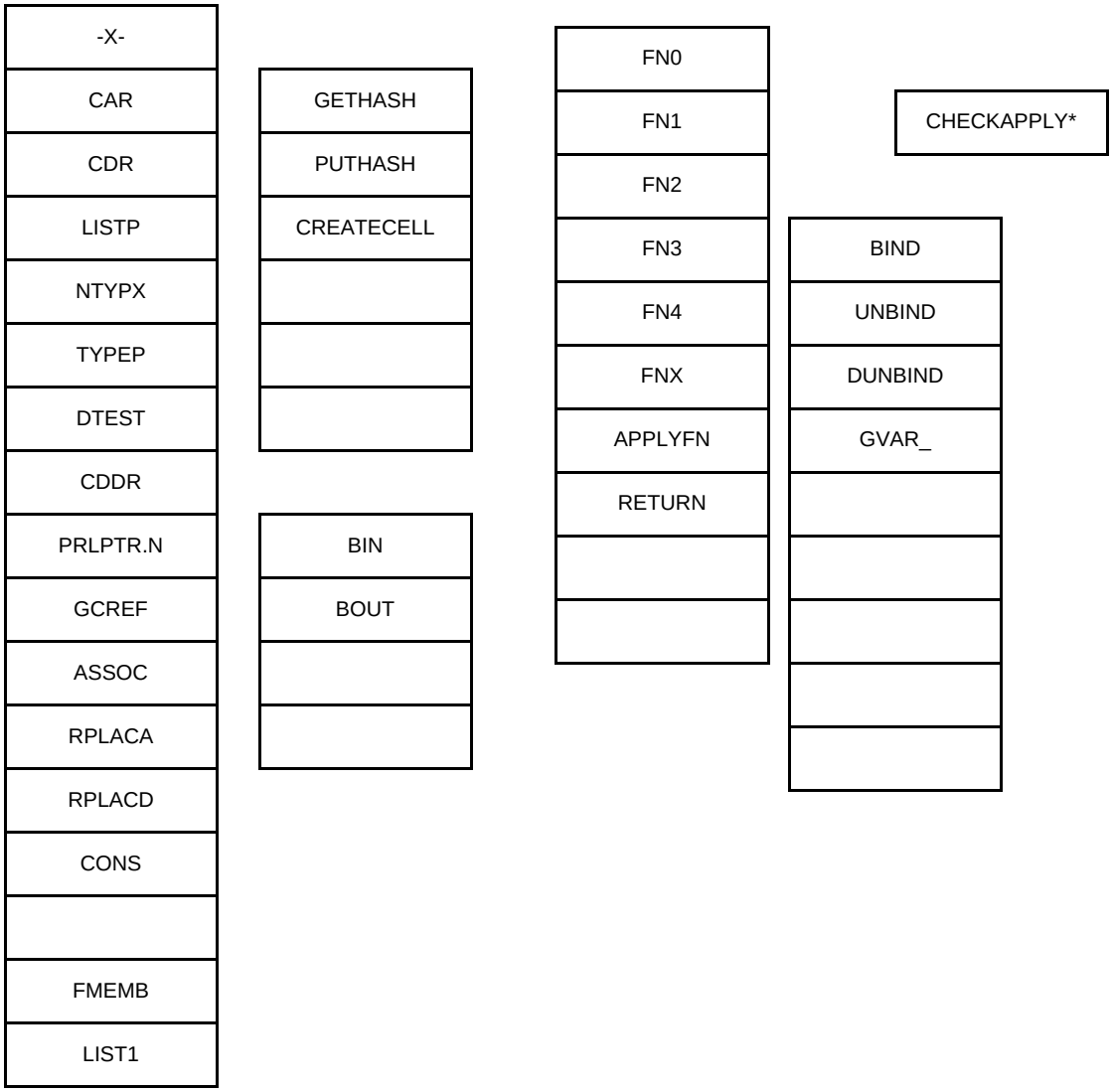
01 : CDR is PREV

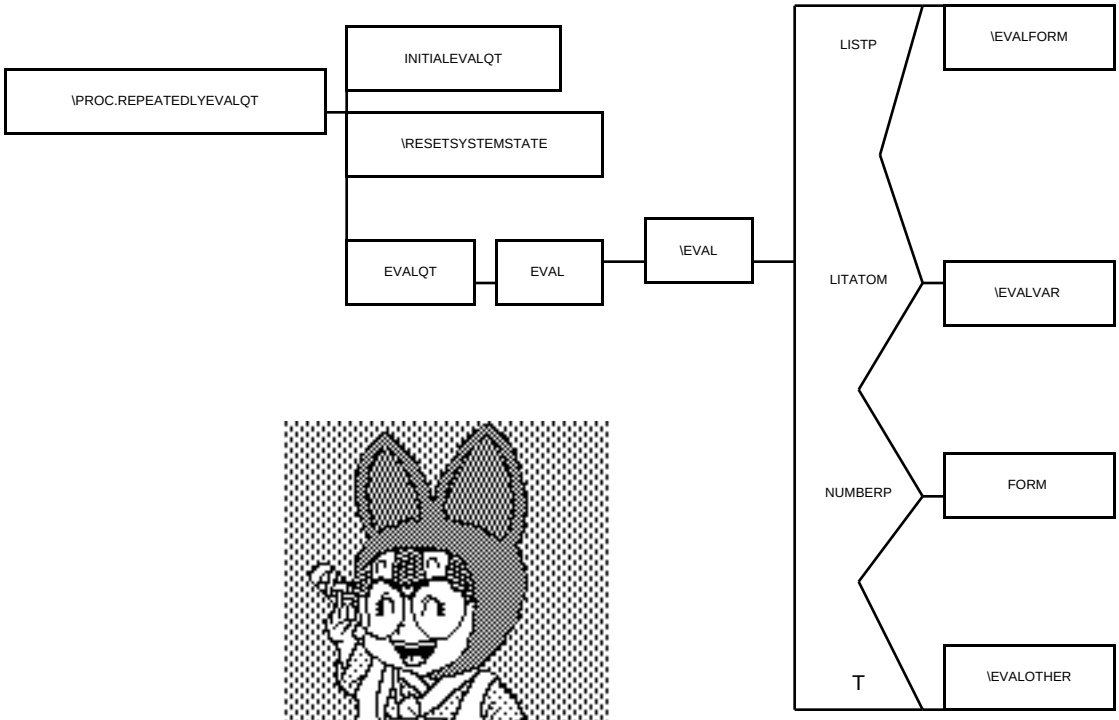
10 : CDR is NEXT

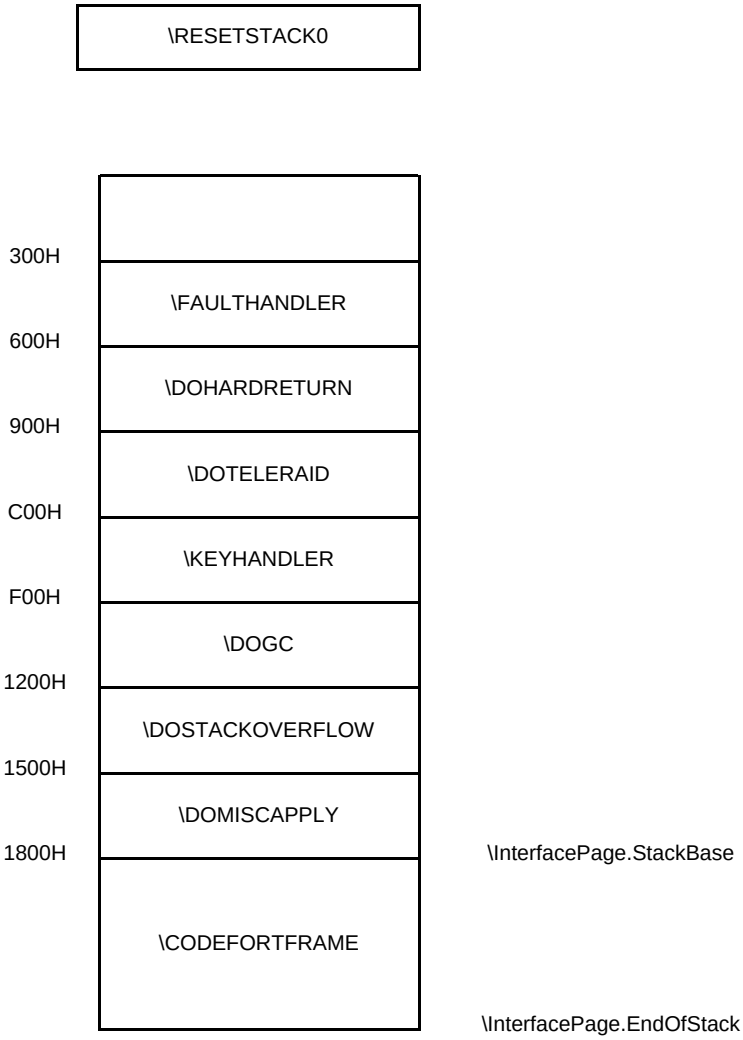
11 : CDR INDIRECT



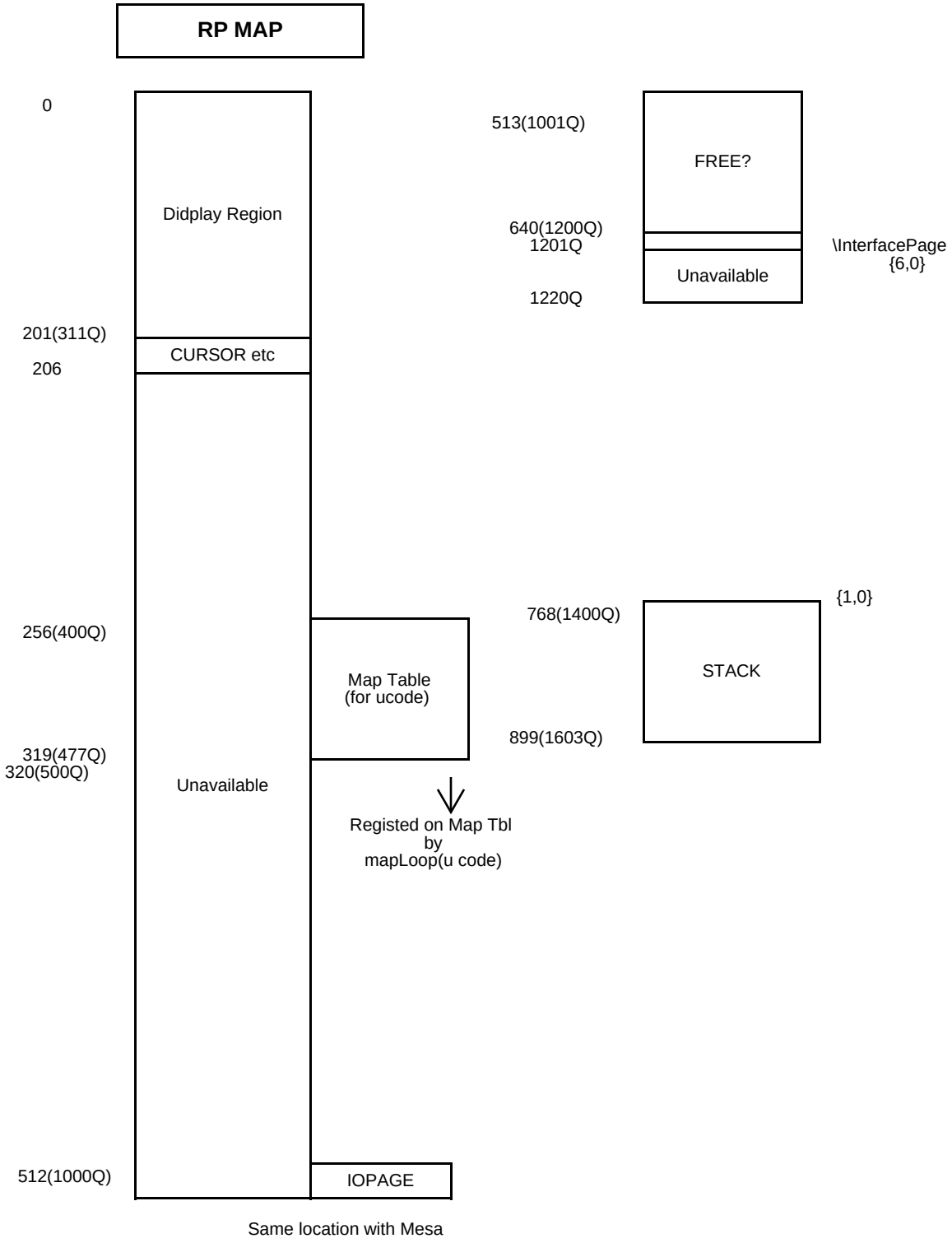
1000	1	1	1024
1004			
1008			
1016			
1020			
1024	1	0	CAR ptr
1028			CDR ptr
1032			

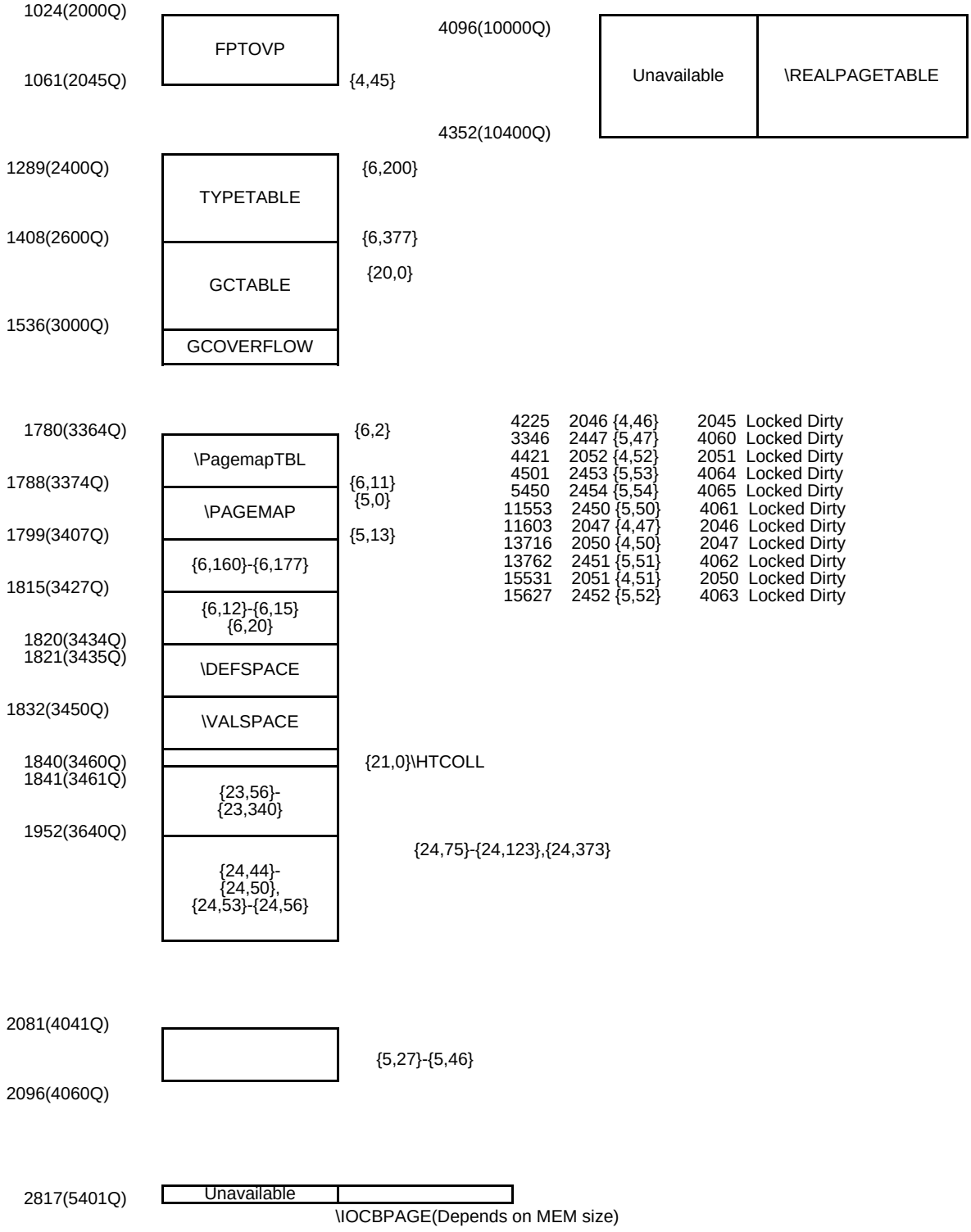

-X-
CAR
CDR
LISTP
NTYPX
TYPEP
DTEST
CDDR
PRLPTR.N
GCREF
ASSOC
RPLACA
RPLACD
CONS
FMEMB
LIST1
GETHASH
PUTHASH
CREATECELL
BIN
BOUT
FN0
FN1
FN2
FN3
FN4
FNX
APPLYFN
RETURN
CHECKAPPLY*
BIND
UNBIND
DUNBIND
GVAR_

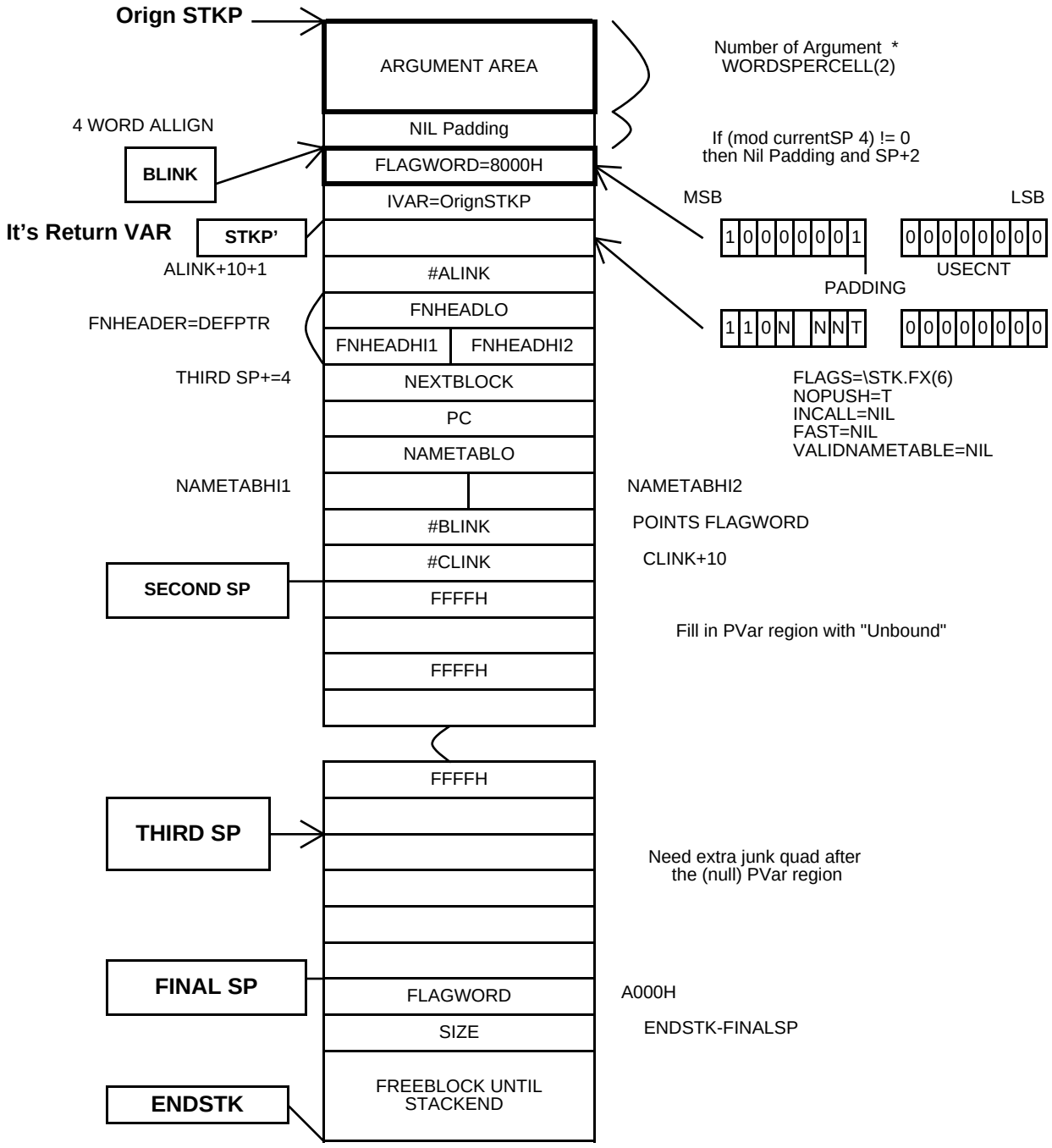

\PROC.REPEATEDLYEVALQT
INITIALEVALQT
\RESETSYSTEMSTATE
EVALQT
EVAL
\EVAL
LISTP
\EVALFORM
LITATOM
\EVALVAR
NUMBERP
FORM
T
\EVALOTHER



\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)
TYPETABLE
{6,200}
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}
2081(4041Q)
{5,27}-{5,46}
2096(4060Q)
2817(5401Q)
Unavailable
\IOCBPAGE(Depends on MEM size)
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


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
FNHEADER=DEFPTR
FNHEADLO
1 1 0 N N N T
0 0 0 0 0 0 0 0
FNHEADHI1
FNHEADHI2
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
FREEBLOCK UNTIL STACKEND
ENDSTK

FUNC NAME	\LOOKUPPAGEMAP

ID	

CALLING SEQUANCE

(\LOOKUUPPAGEMAP VP)

LOCAL VAL

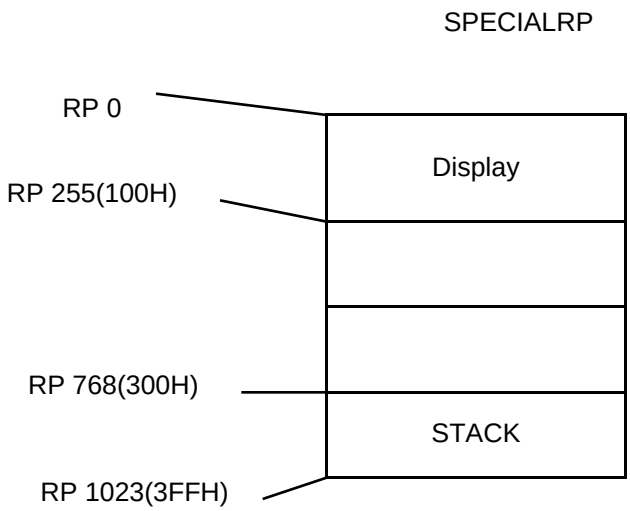
PRIMENTRY ** (\GETBASE \PageMapTBL (fetch (VP PRIMARYKEY) of VP]

GLOOVAL VAL

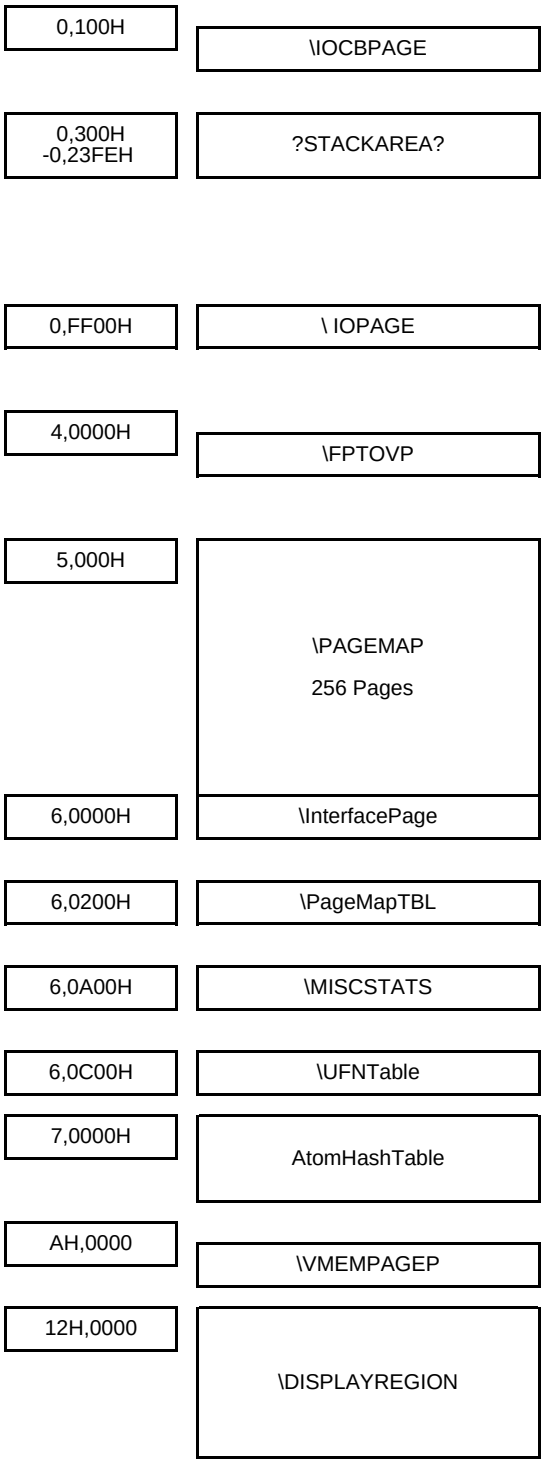
NAME	TYPE
\PageMapTBL (R) \EMPTYPMTENTRY (R) \PAGEMAP (R) VP (R)	RECORD VP

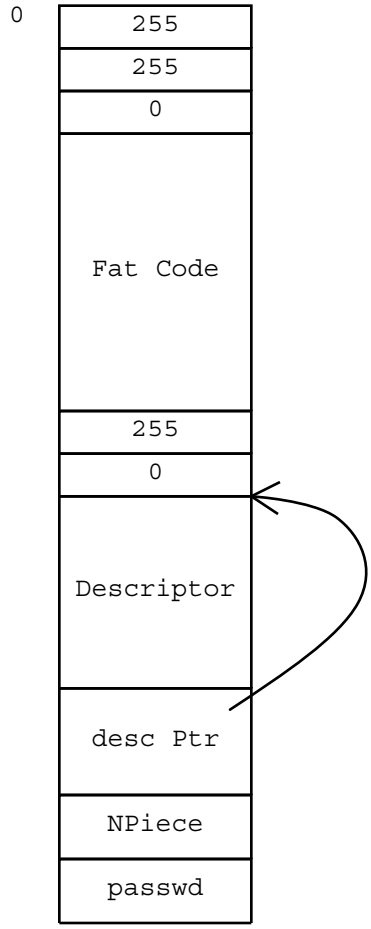
CALL FUNCTIONS

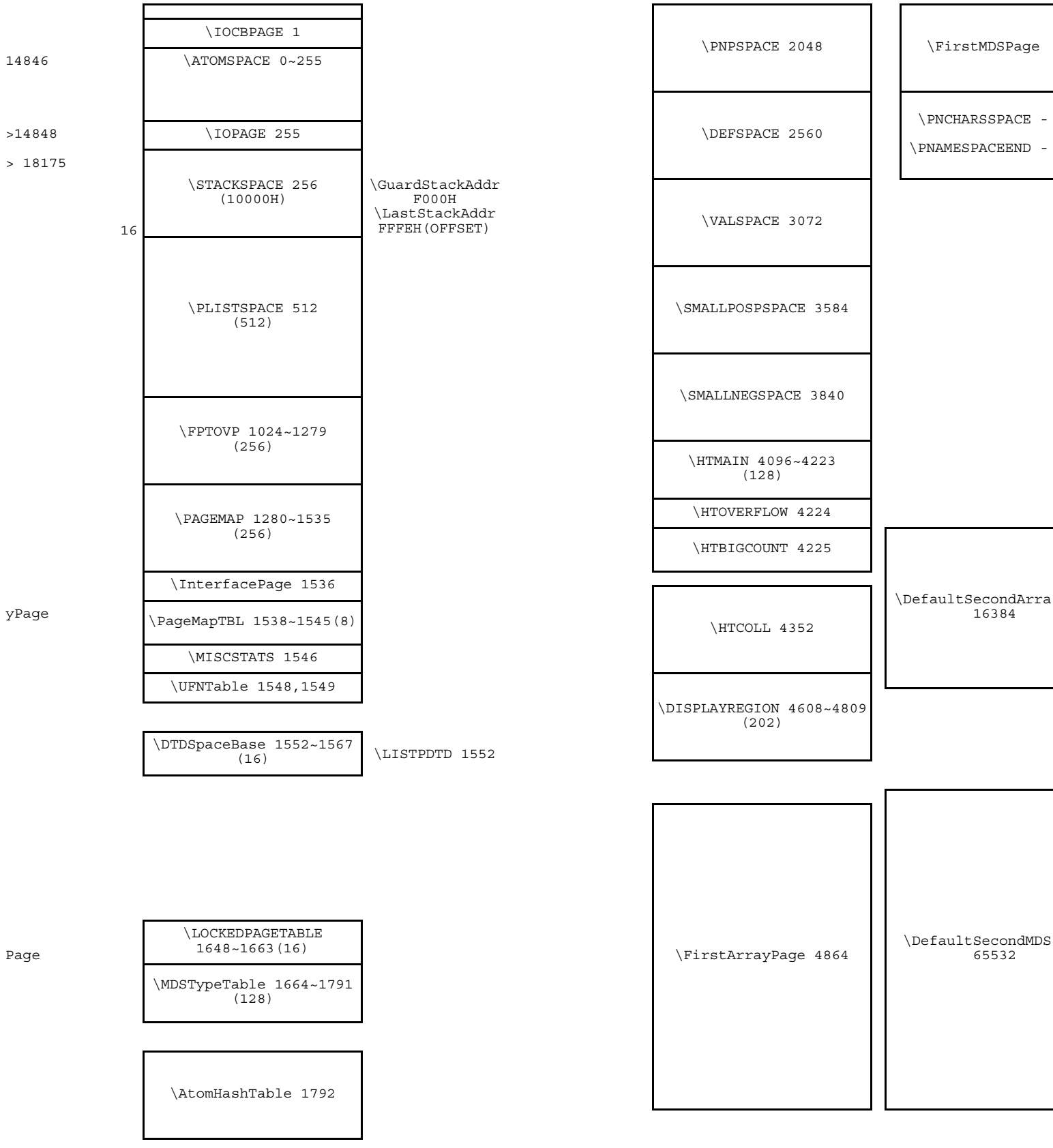
\GETBASE

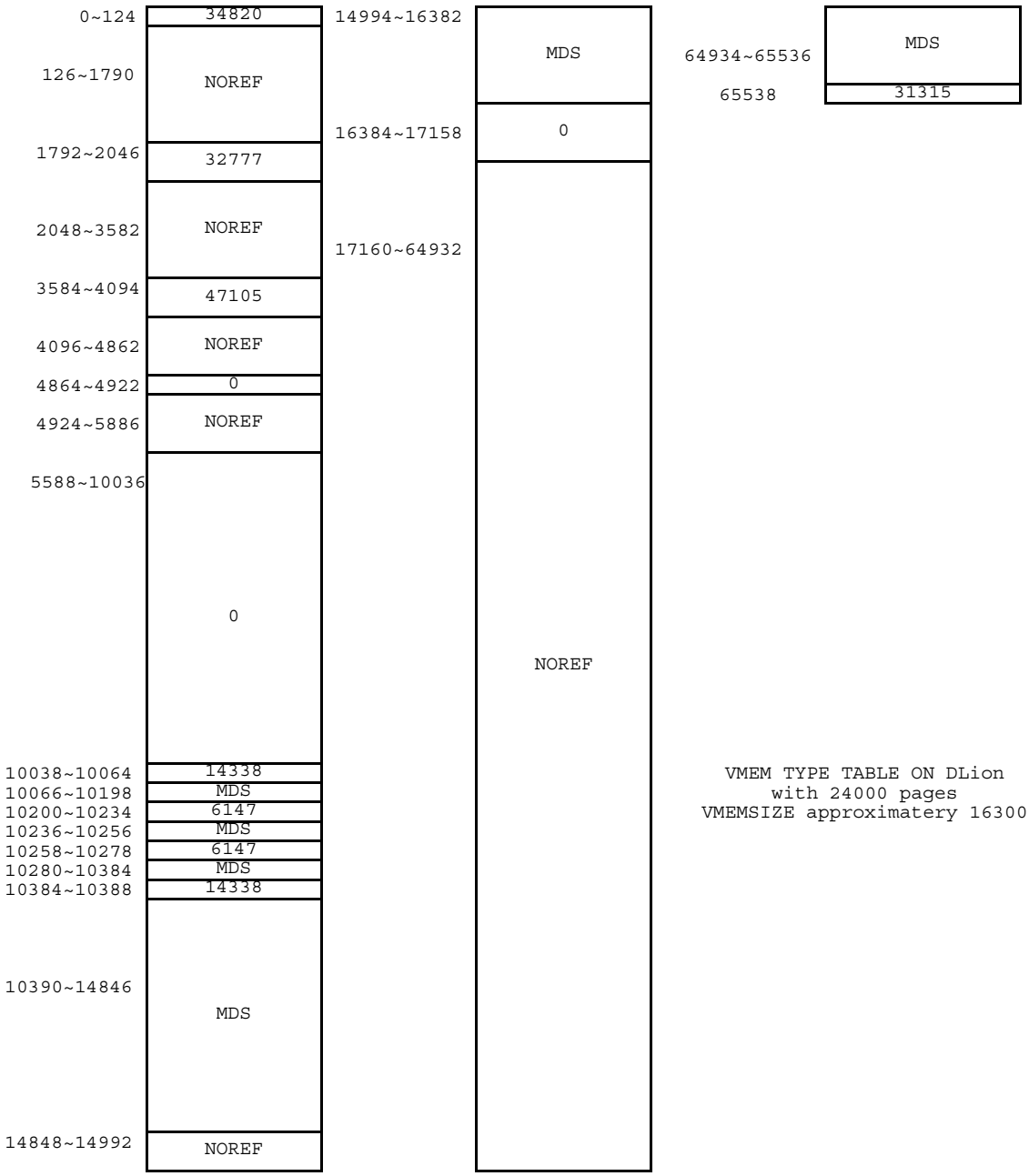

SPECIALRP
RP 0
Display
RP 255(100H)
RP 768(300H)
STACK
RP 1023(3FFH)

Memory Stats on STARTUP




0
255
255
0
Fat Code
255
0
Descriptor
desc Ptr
NPiece
passwd


\IOCBPAGE 1
14846
\ATOMSPACE 0~255
>14848
\IOPAGE 255
> 18175
\STACKSPACE 256
(10000H)
\GuardStackAddr
F000H
\LastStackAddr
FFFEH(OFFSET)
16
\PLISTSPACE 512
(512)
\FPTOVP 1024~1279
(256)
\PAGEMAP 1280~1535
(256)
\InterfacePage 1536
yPage
\PageMapTBL 1538~1545(8)
\MISCSTATS 1546
\UFNTable 1548,1549
\DTDSpaceBase 1552~1567
(16)
\LISTPDTD 1552
Page
\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



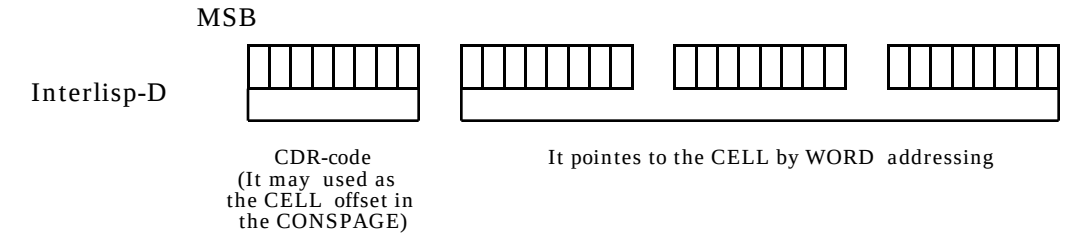
VMEM TYPE TABLE ON DLion
with 24000 pages
VMEMSIZE approximatery 16300

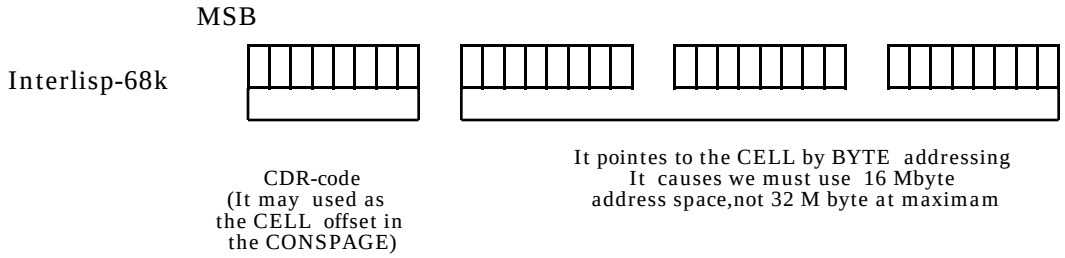
Spec of \ATOMSPACE

0	NIL ptr
1	NOBIND ptr
	ASCII code from 0 to 47, and 58 - 255
367Q	
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

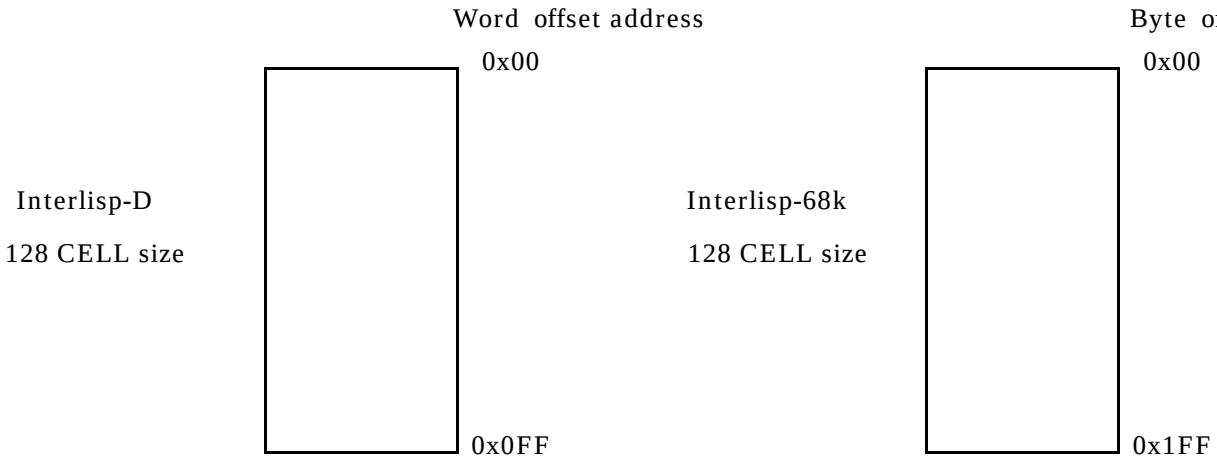
\PTRHUNK16
\PTRHUNK24
\PTRHUNK32
\PTRHUNK42
\PTRHUNK64
\UNBOXEDHUNK1

* CELL

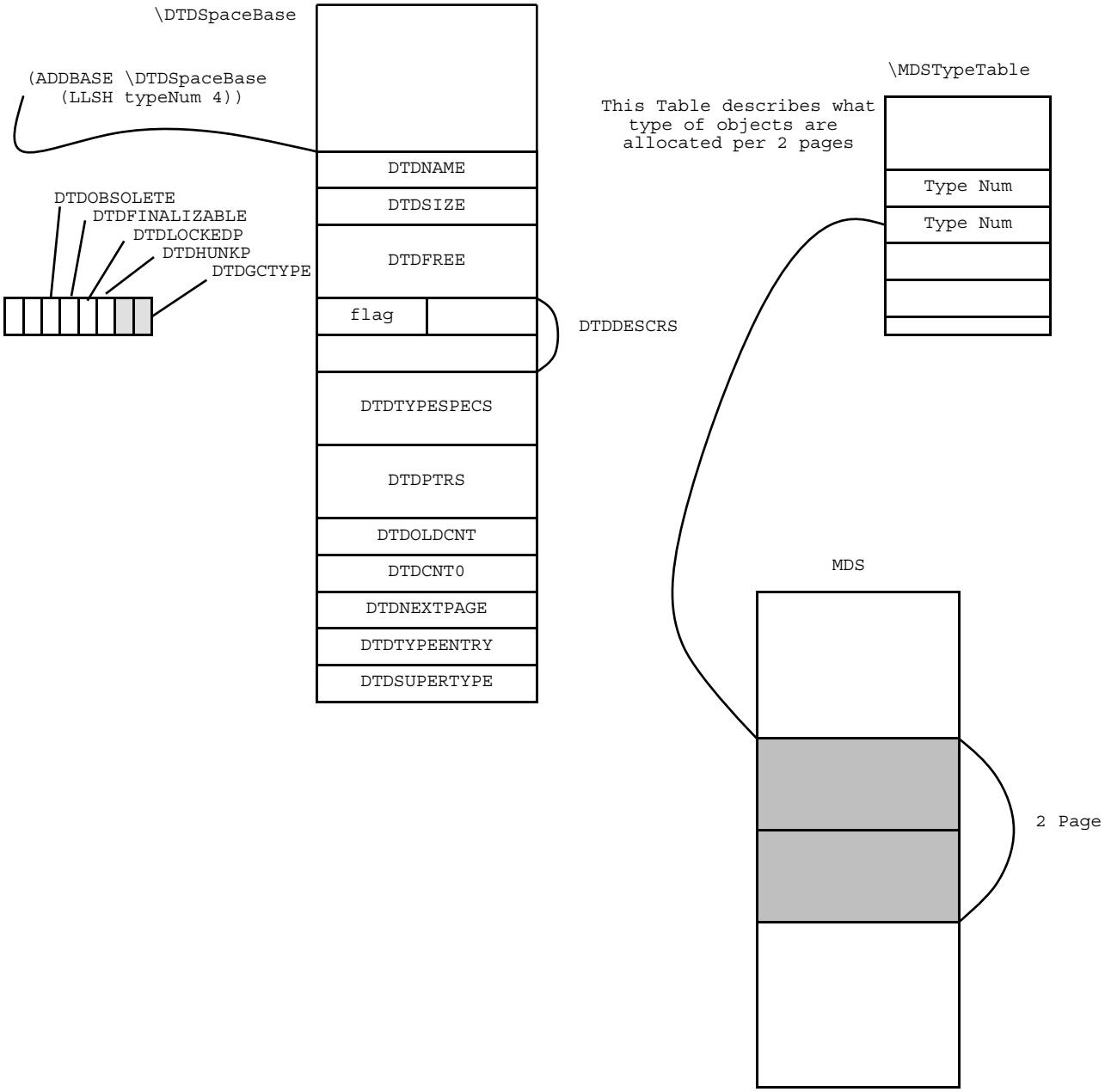


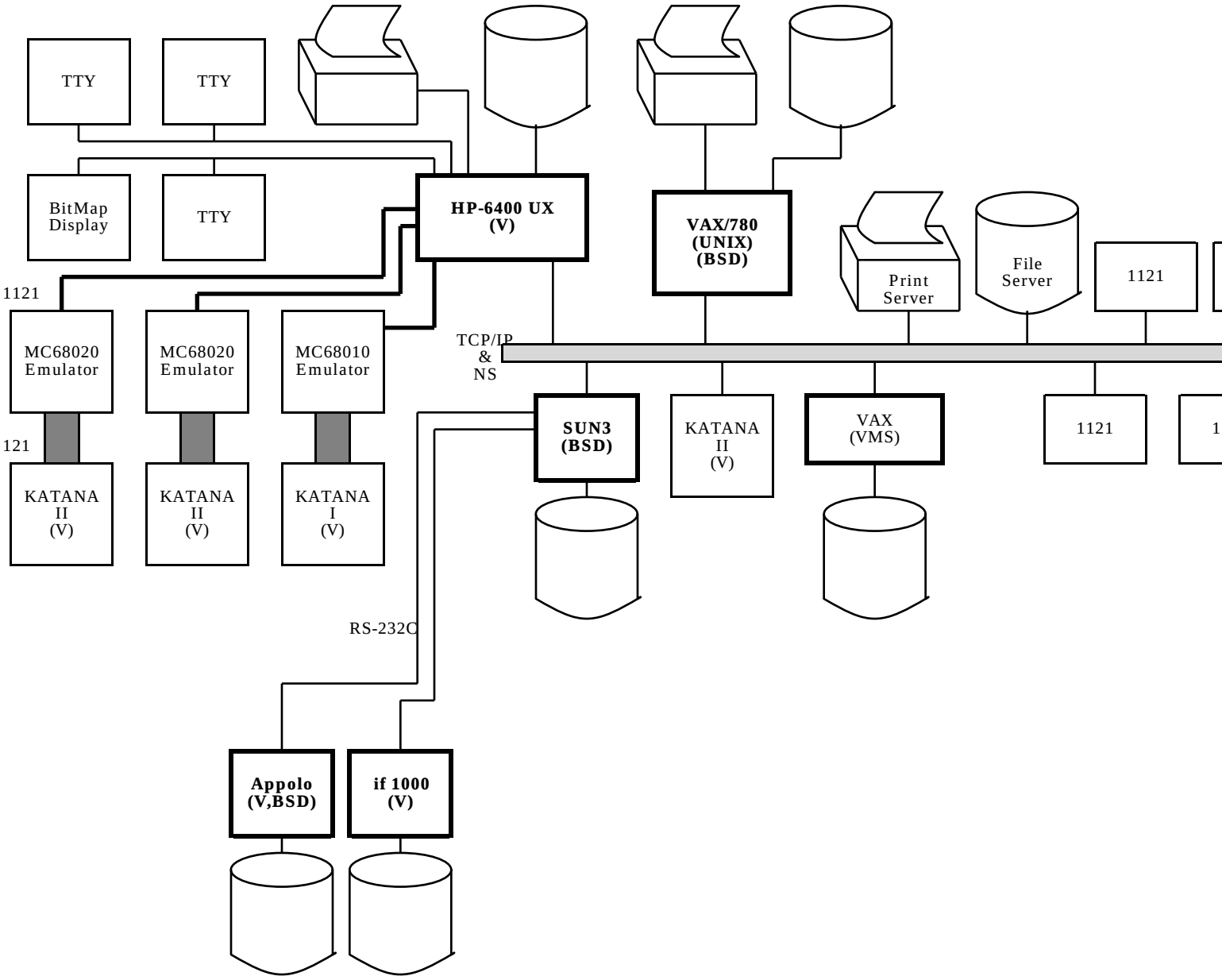


* CONSPAGE



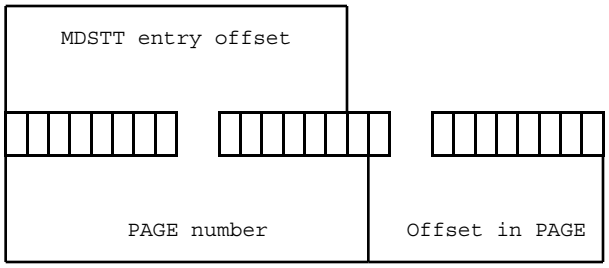
: 1 Cell 32 bit width, and !CONSPAGE includes 128 Cells


\DTDSpaceBase
(ADDBASE \DTDSpaceBase
(LLSH typeNum 4))
DTDNAME
DTDSIZE
DTDFREE
DTDOBSOLETE
DTDFINALIZABLE
DTDLOCKEDP
DTDHUNKP
DTDGCTYPE
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

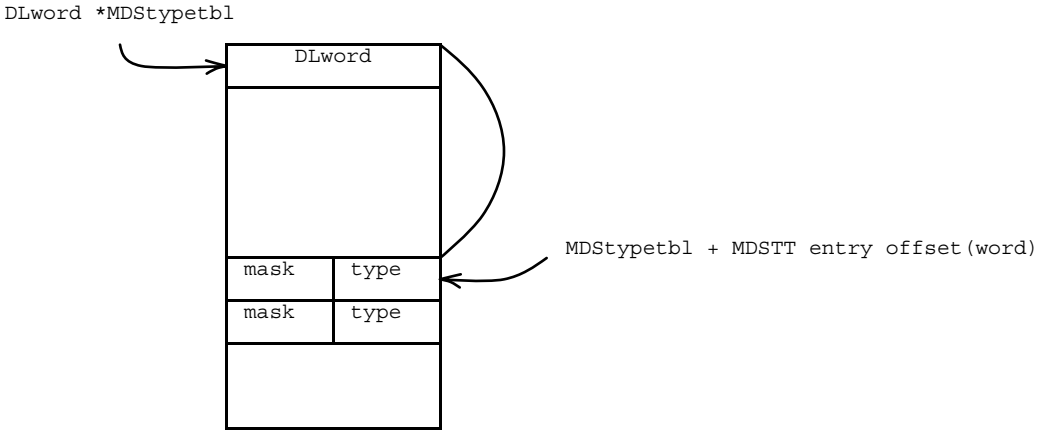
by Takeshi Shimizu



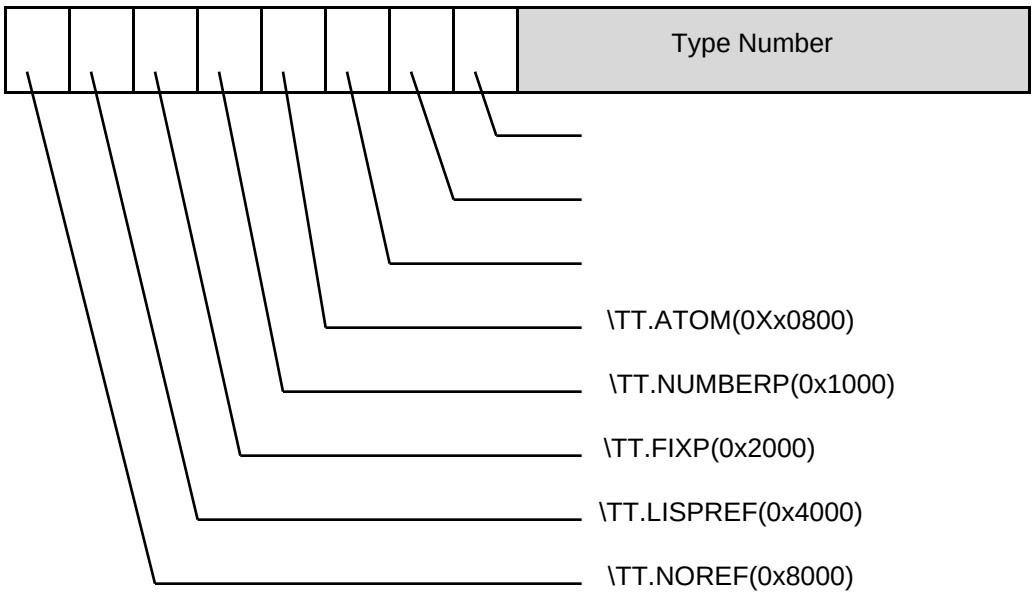
1 Page = 512 byte = 9 bit = 0x1FF

1 MDS entruy explains the type of 2 pages -- 10 bit

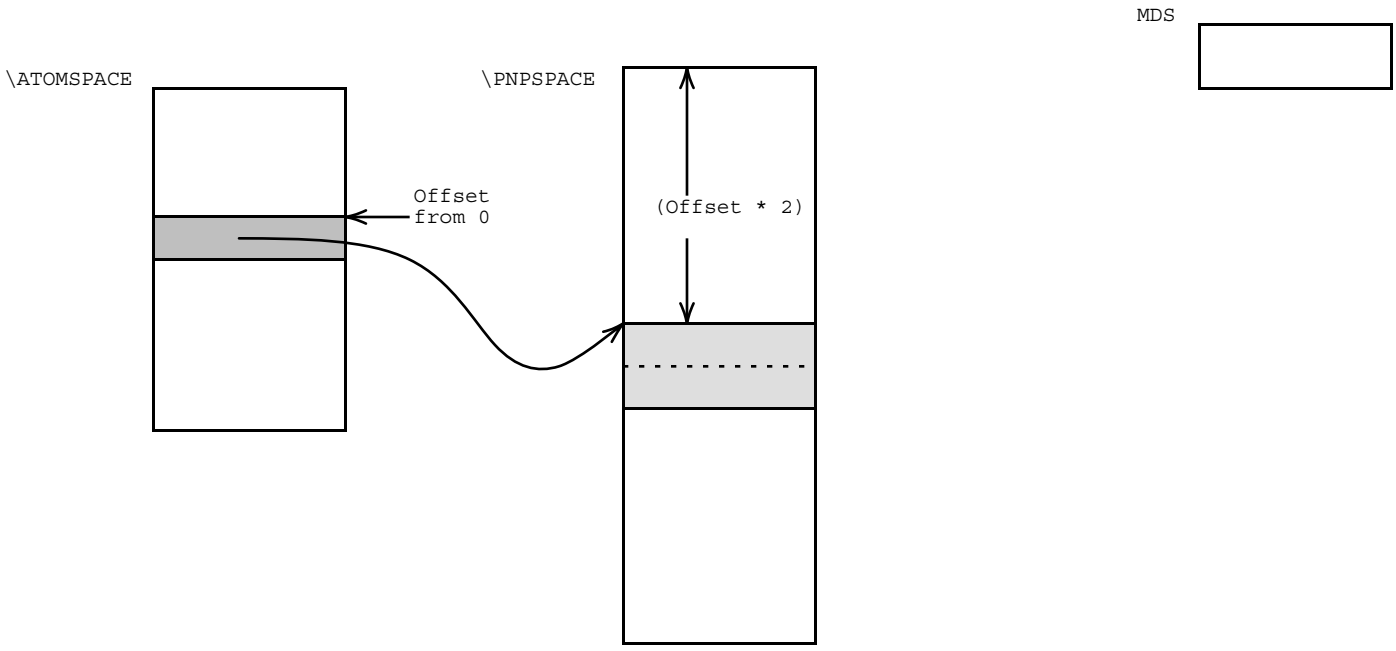
Method of accessing to MDSTT

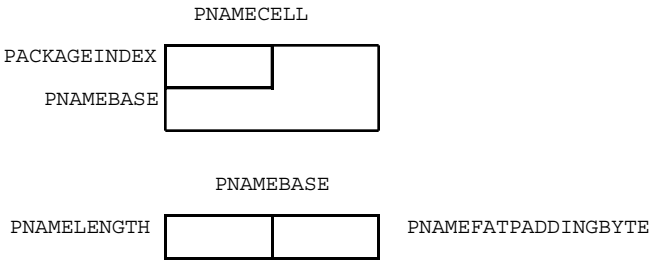


MDSTtpeTable Entry



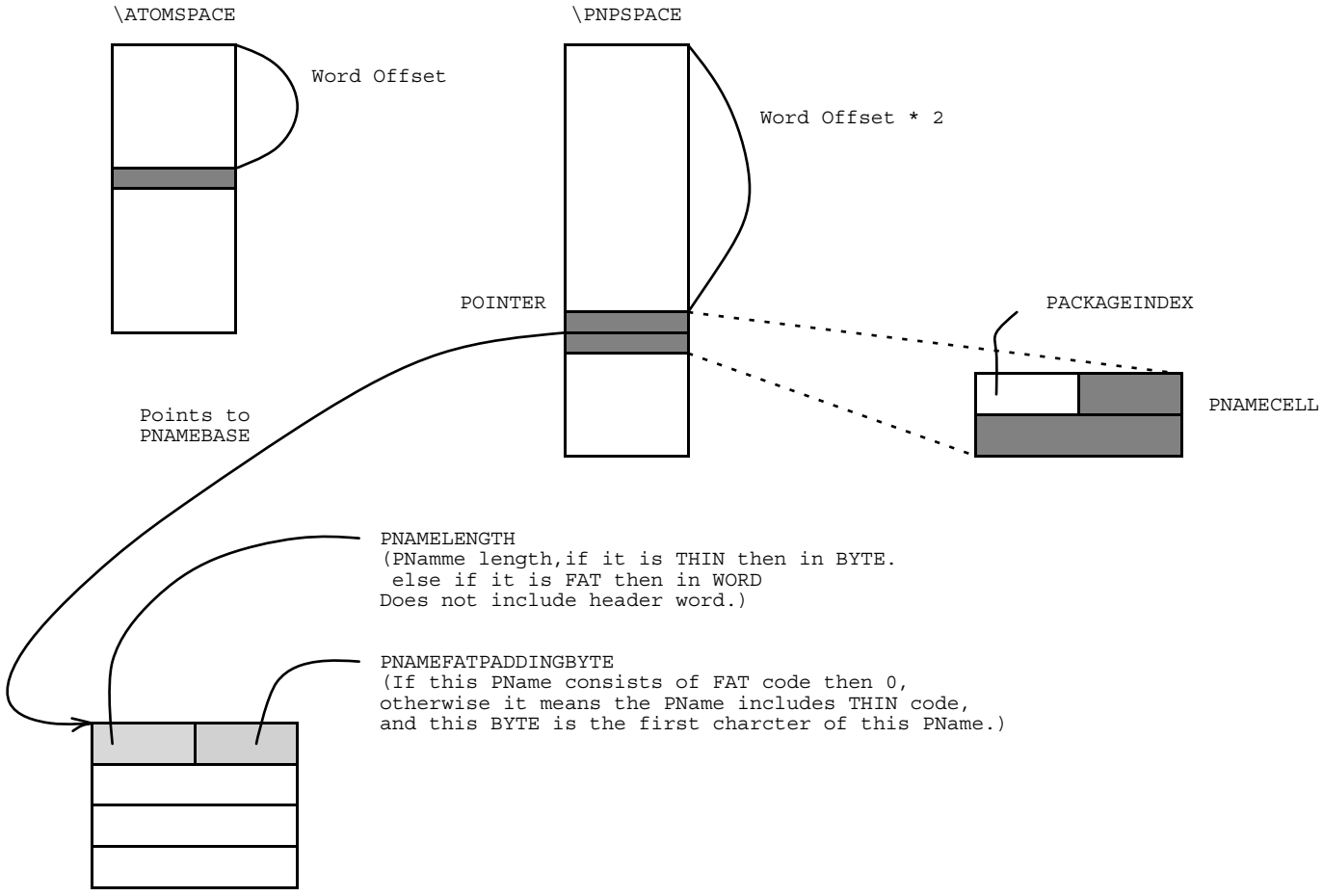
How to Access PName





How to Access PName

by Takeshi Shimizuu



Example: NIL

Its index number is 0.
And,Its Length is 3 bytes.

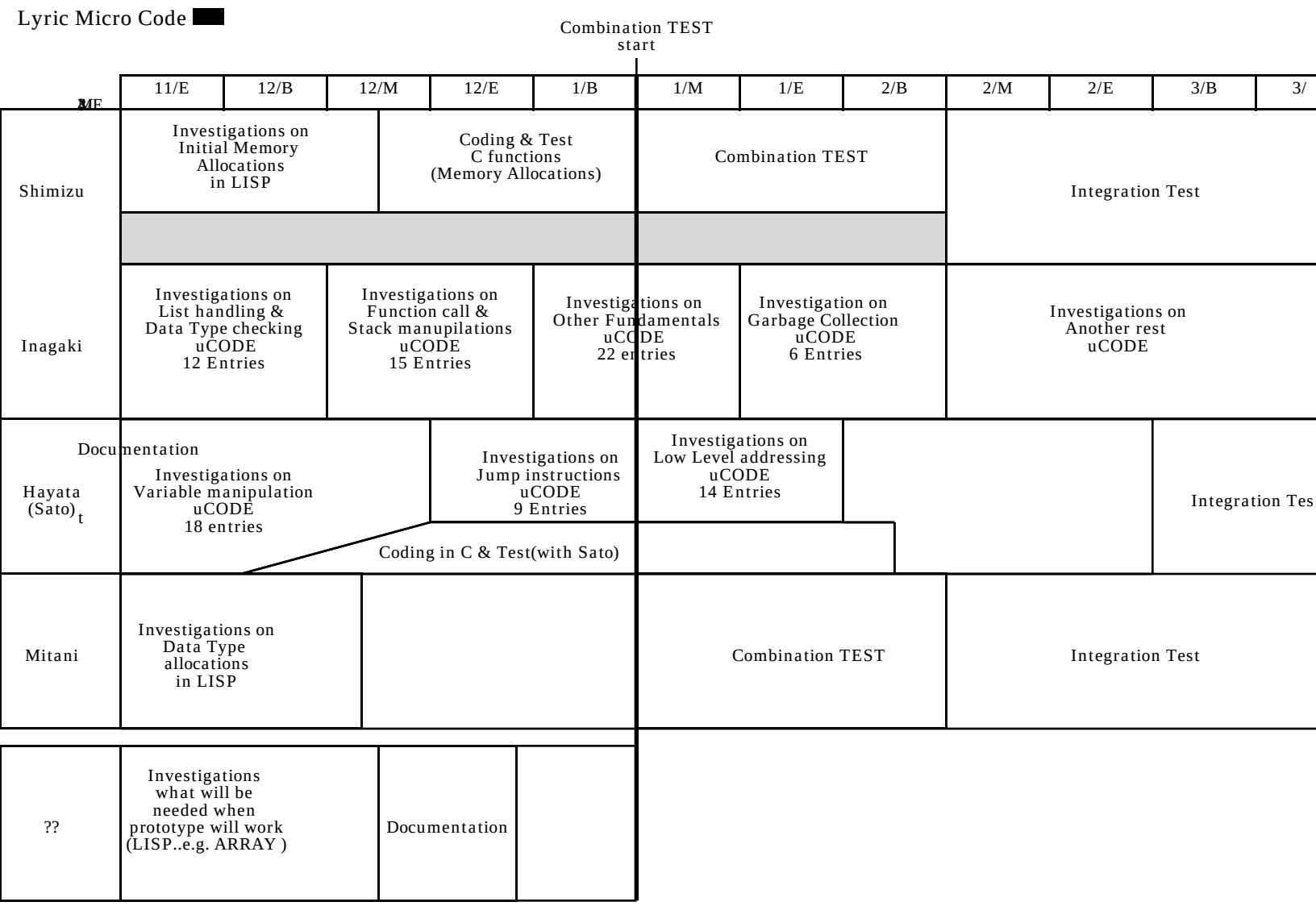
3	N
I	L

Example

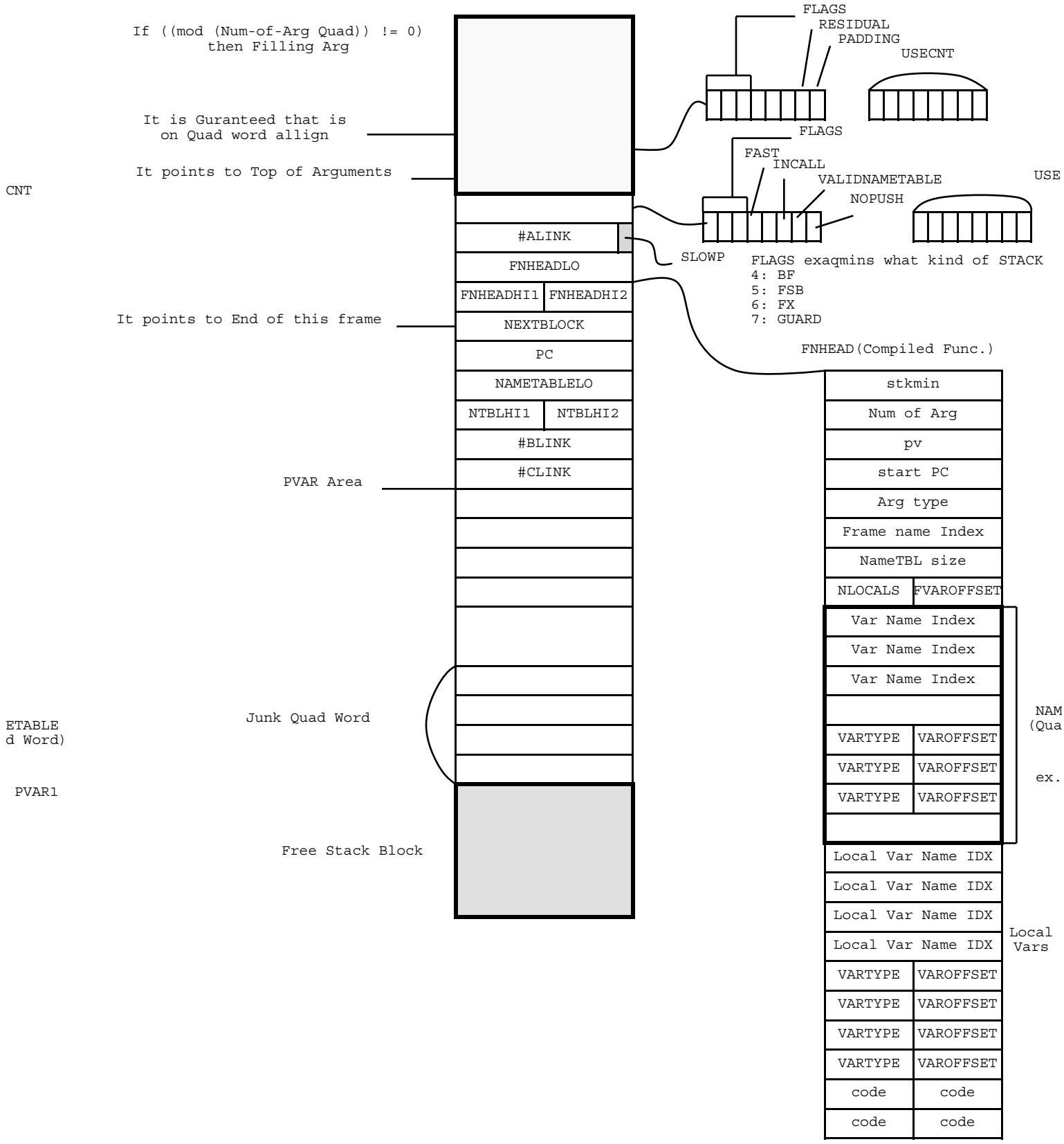
Example: ■

Its index number is X.
And,Its Length is 3 wordes.

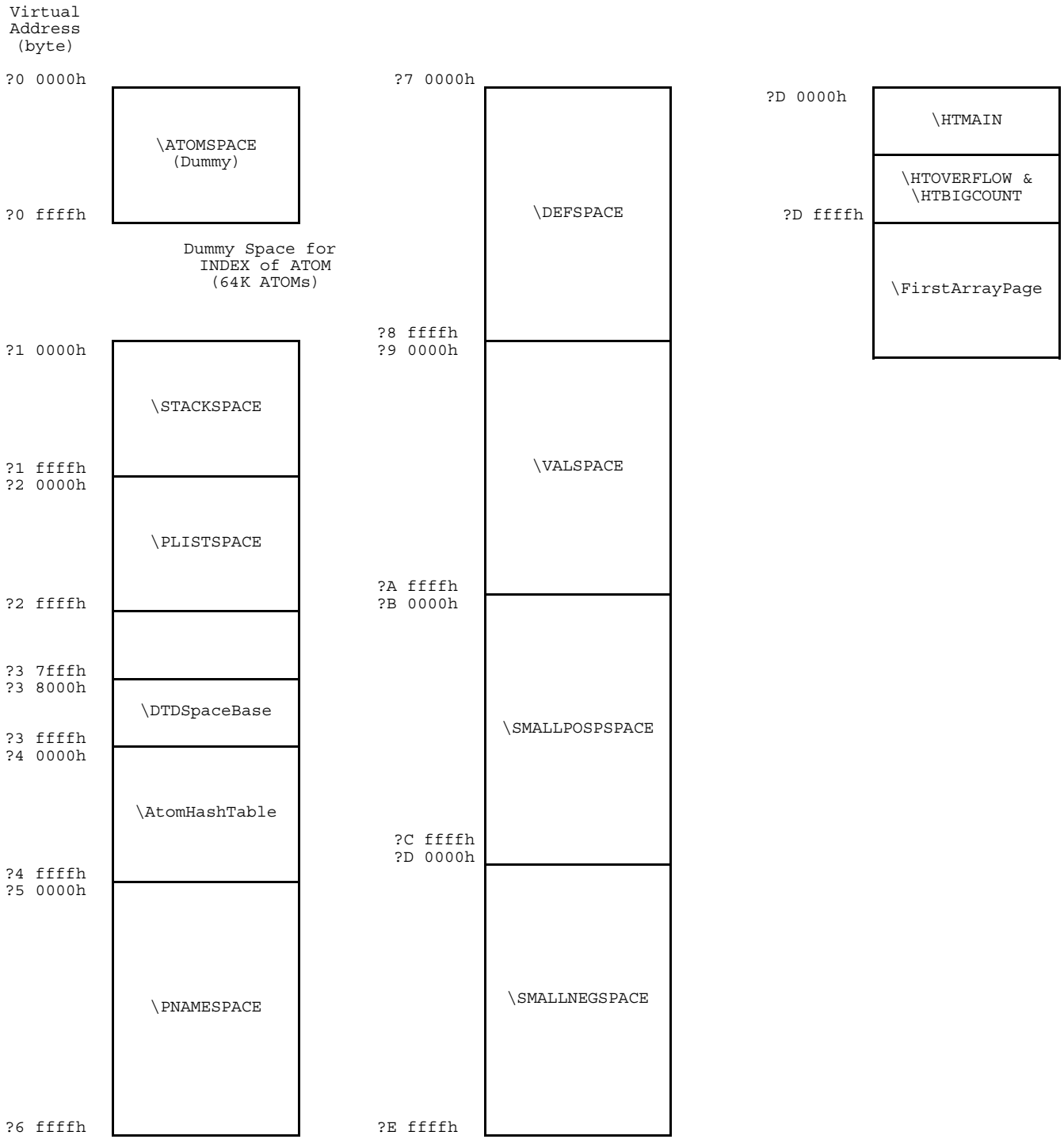
3	0
■	
■	
■	


Lyric Micro Code
Combination TEST start
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)
Documentation
Investigations on Variable manipulation uCODE 18 entries
Investigations on Jump instructions uCODE 9 Entries
Investigations on Low Level addressing uCODE 14 Entries
Coding in C & Test(with Sato)
Integration Tes
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

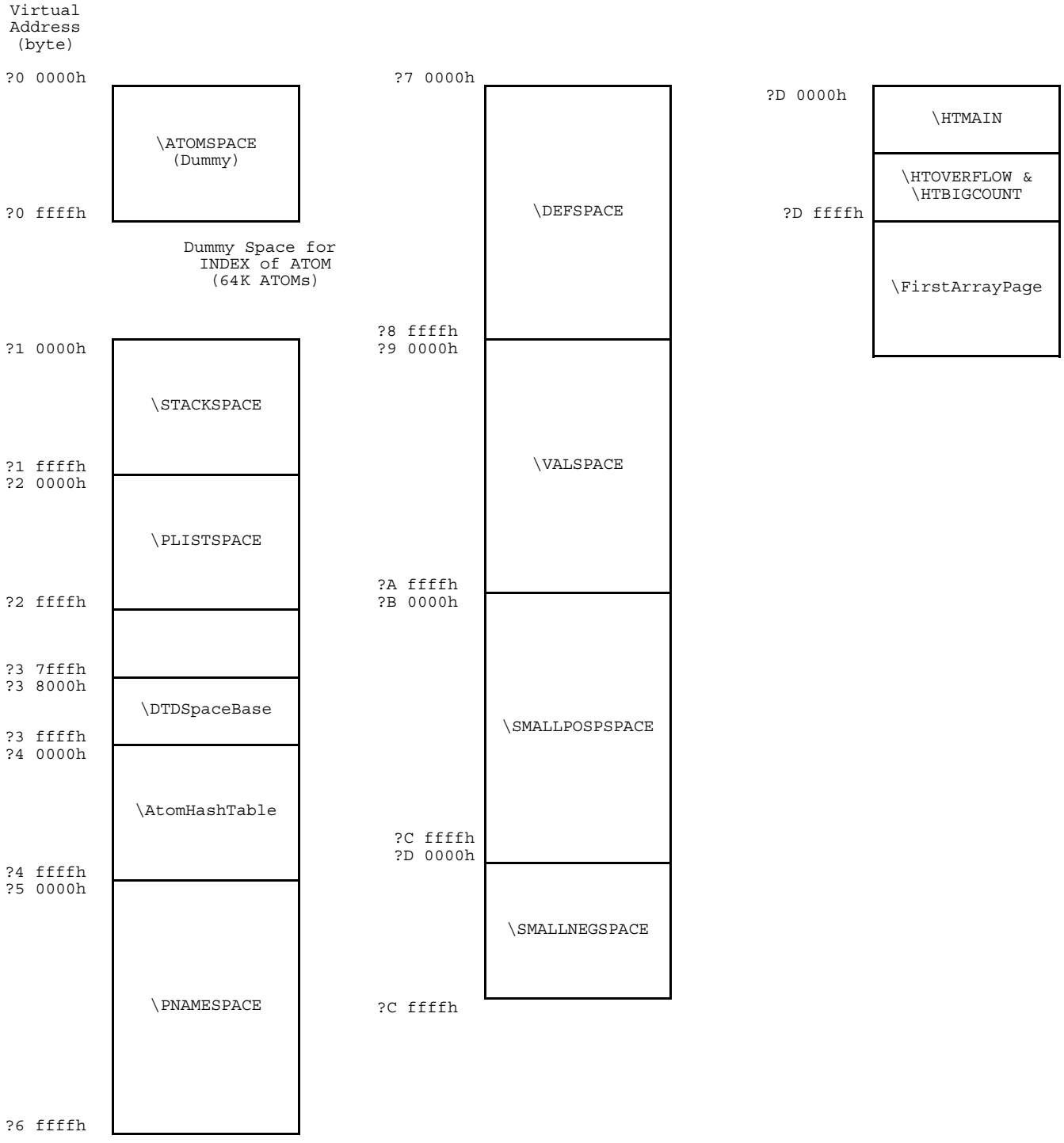


Memory Map(for TEST on SUN-3)



(ADDBASE (LLSH ATOMINEX 1) \PNAMESPACE)

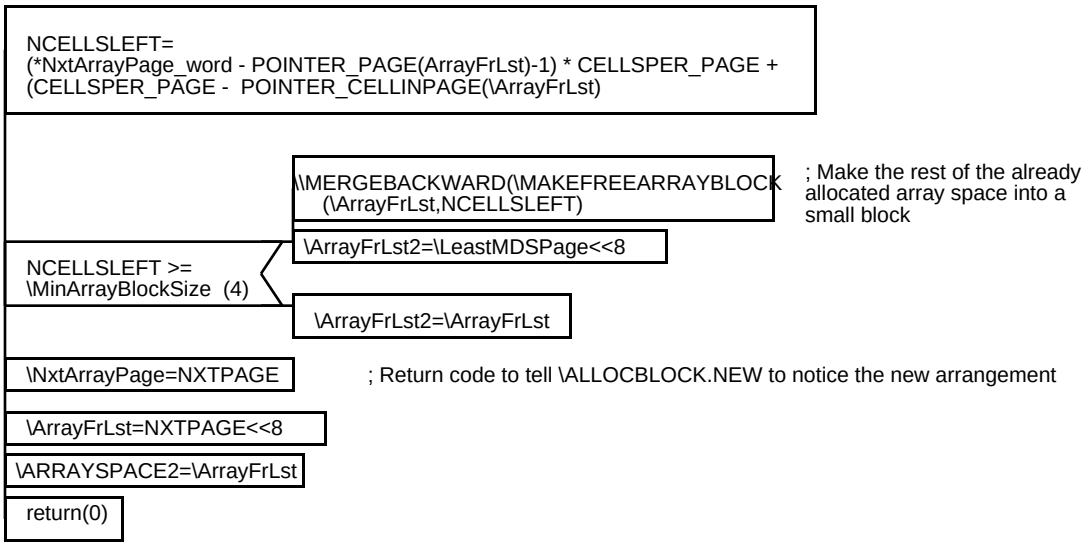
Memory Map(for TEST on SUN-3)



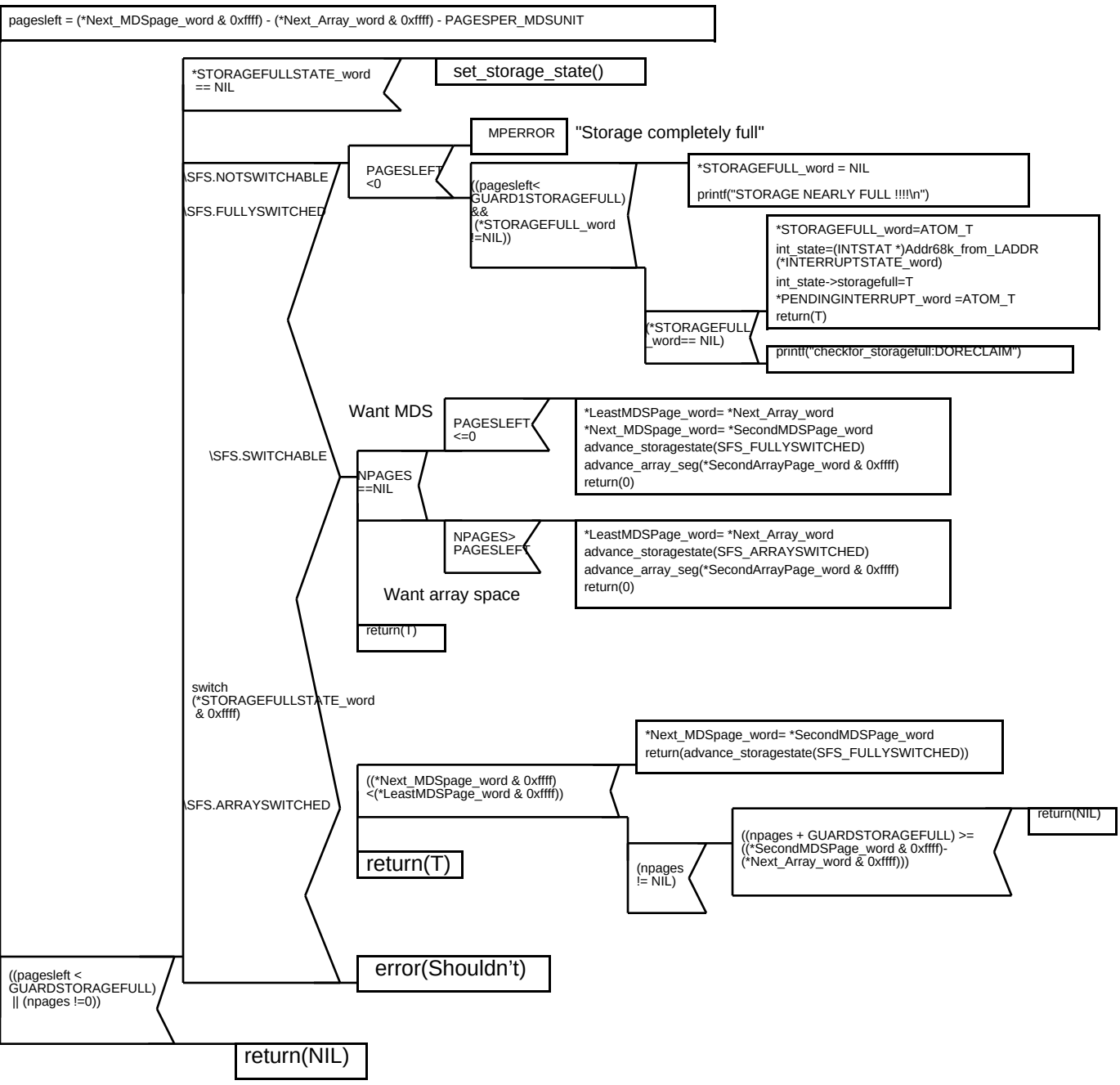
(ADDBASE (LLSH ATOMINEX 1) \PNAMESPACE)

\ADVANCE_ARRAY_SEG(NXTPAGE)

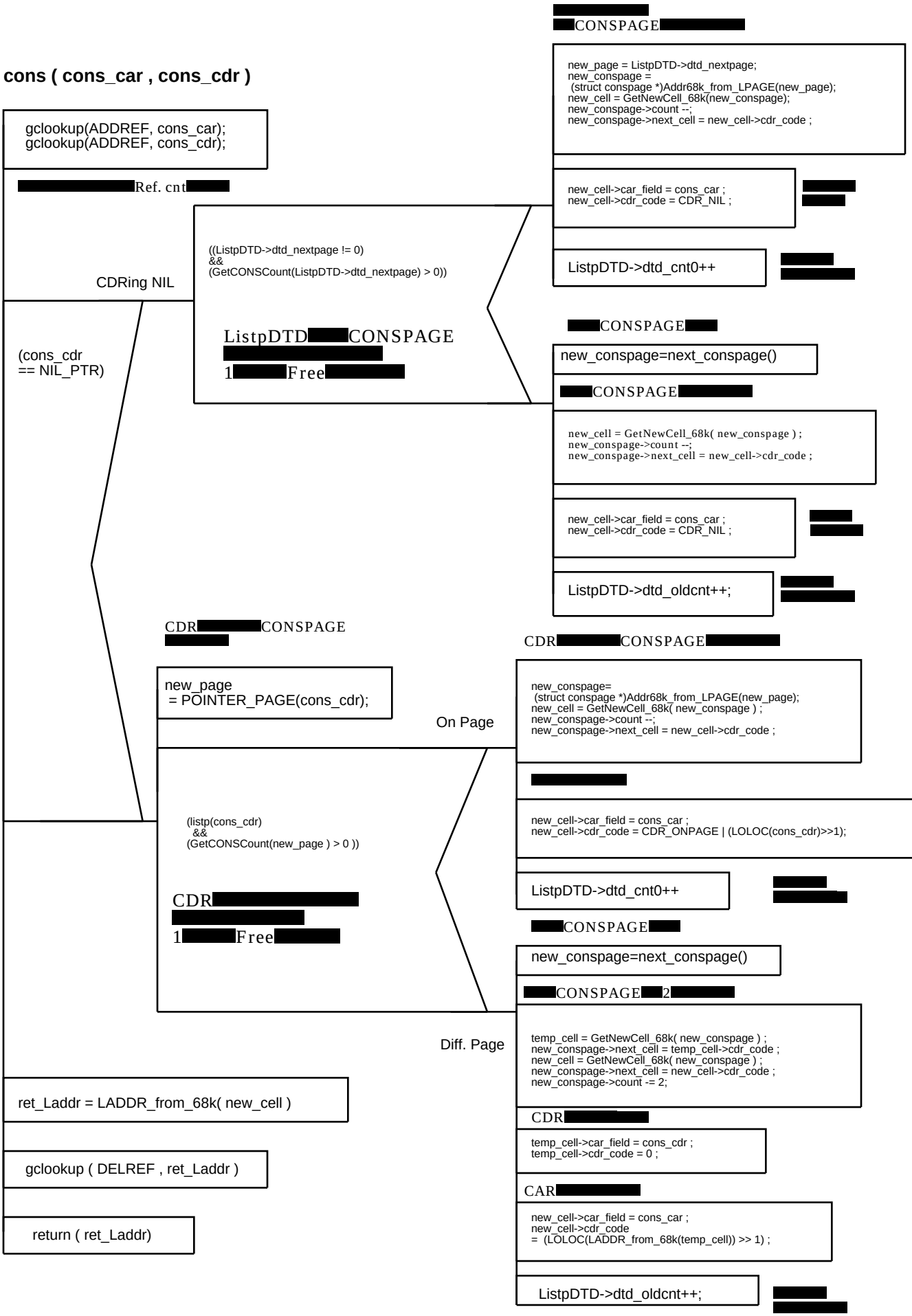
Called when the first 8mb are exhausted, and we want to switch array space into the next area, starting with page NXTPAGE -- have to first clean up what's left in the old area



checkfor_storagefull(npages)



cons (cons_car , cons_car)



OP_contextswitch

PushCStack

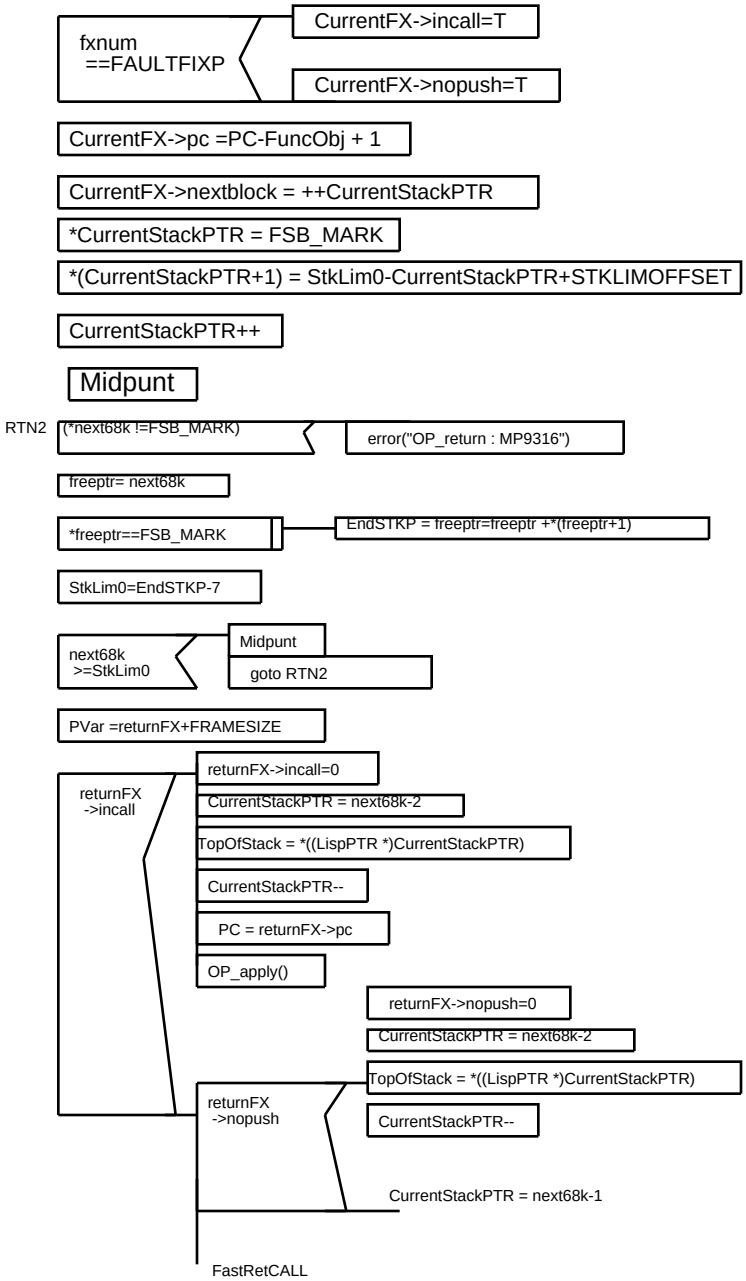
contextswitch(TopOfStack &0xffff)

CurrentFX =STK_OFFSET | Addr68k_from_LADDR(*(InterfacePage + fxnum))

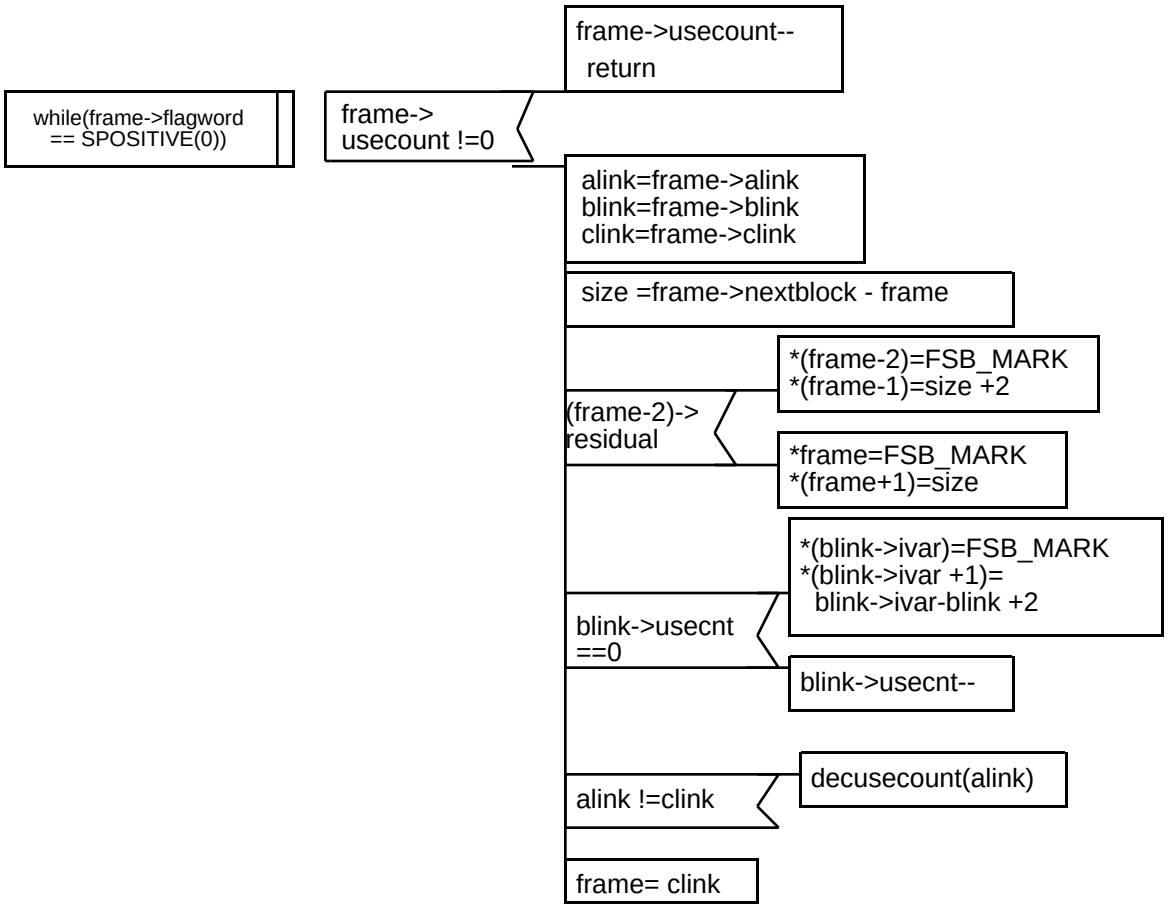
*(InterfacePage + fxnum) = LOLOC(LADDR_from_68k(CurrentFX))

Midpunt

contextswitch(fxnum)



DECUSECOUNT(FRAME)



```
SETCLINK(fx,val)  {  fx->clink=val+FRAMESIZE;
  if(!(fx->alink & 1)) {  fx->blink=fx-DLWORDSPER_CELL;
    fx->alink |= 1;}
```

```
SETACLINK(fx,val)  { if(!(fx->alink&1))fx->blink=fx-DLWORDSPER_CELL;
  fx->clink=val+FRAMESIZE;
  fx->alink=fx->clink+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)

hframe->framename ==\CONTEXTSWITCH

hrframe=hrframe->clink

returner= hrframe->clink

Checking whether this frame is HARDRETURN or not

returnee = retuner->clink

checkFX(retunee) && checkFX(retuner) &&(returner- >slowp) &&
((retuner->alink & 0xfffe) != retunee) ||

retunee != (ax=(retuner->alink&0xfffe))

SETALINK(retuner->alink,retunee)decusecount(ax)

goto DOIT

retunee->
usecount !=0

ax= retunee->nextblock

for((new=ax+ax->size);new->flags==STK_FSB;
new=ax+ax->size)

ax->size += new->size

ax->flags
==STK_FSB

ax->size <= MINEXTRASTACKWORDS

goto DOIT

goto LAST

oldax=ax
ax=GETBLINK(retuner)

goto DOIT

oldax==ax->ivar

ax->usecnt !=0

goto DOIT

DOIT

size=(retunee->nextblock -retunee)

goto DOIT

(retunee->validnametable)&&
((retunee->nametablehi2)==STK.HI

nametbl_on_stk=T

align =(retunee-DLWORDSPER_CELL)%
DLWORDSPER_QUAD

new=FREESTACKBLOCK(size,retuner,align)

blt(STK_OFFSET|new, STK_OFFSET | (retunee-DLWORDSPER_CELL)

new->residual=T

new+=DLWORDSPER_CELL

new->nextblock= (new+size)-DLWORDSPER_CELL

retbf= GETBLINK(retunee)

SETBLINK(new, retbf)

new->usecount=0

GETCLINK(fx) (fx->alink &1) ? (fx-clink-FRAMESIZE)
: (fx->alink-FRAMESIZE)

GETALINK(fx) (fx->alink& 0xfffe) -FRAMESIZE

nametbl_on_stack

new->nametblelo +=new-retunee

retnf->usecnt ++

if(!(reutnee->alink &1)) {
 retunee->blink=retunee-2;
retunee->clink=retunee->alink

retunee->alink |=1

incusecount(tmp=GETCLINK(retunee))

tmp !=(ax=GETALINK(retunee))

incusecount((ax)

decusecount(retunee)

SETACLINK(retuner,new)

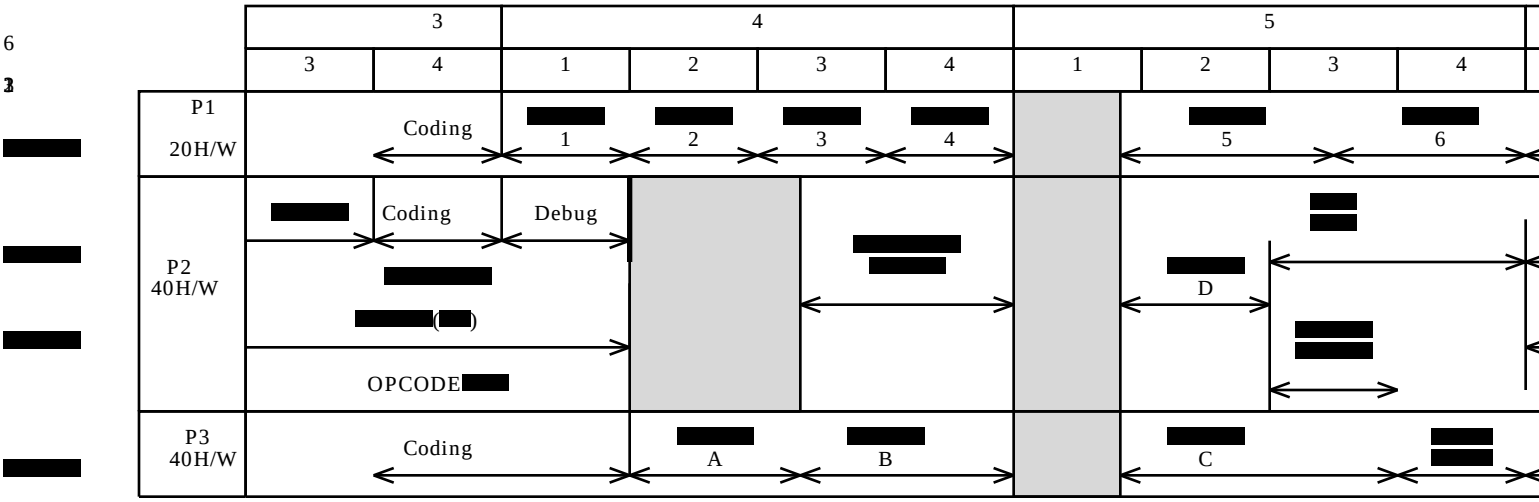
retunee=new

LAST

smashlink(hrframe , retunee ,retunee)

Replanning 24 March 1987 By

Takeshi Shimizu



1 = { vars.c,vars2.c,binds.c,jump.c,arith.c }

2 = { lowlevel.c }

3 = { vars3.c }

4 = { gvar2.c }

5 = { fvar.c }

6 = { gc.c }

= { initdatatype.c , makeatom.c , array.c }

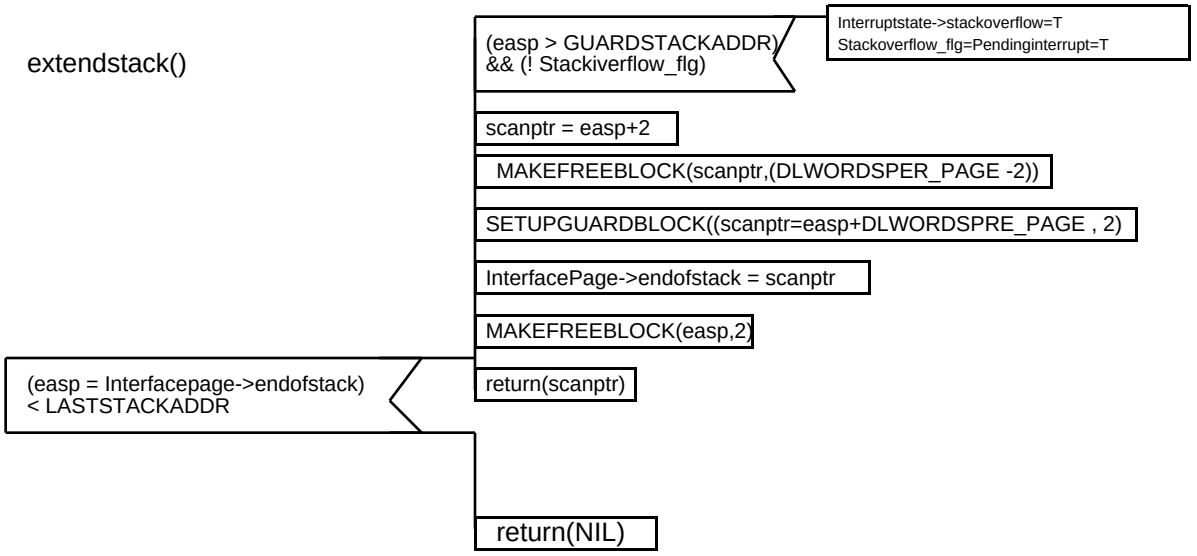
A = { typeof.c, constants.c, OP_eq, OP_copy , OP_atomcellN }

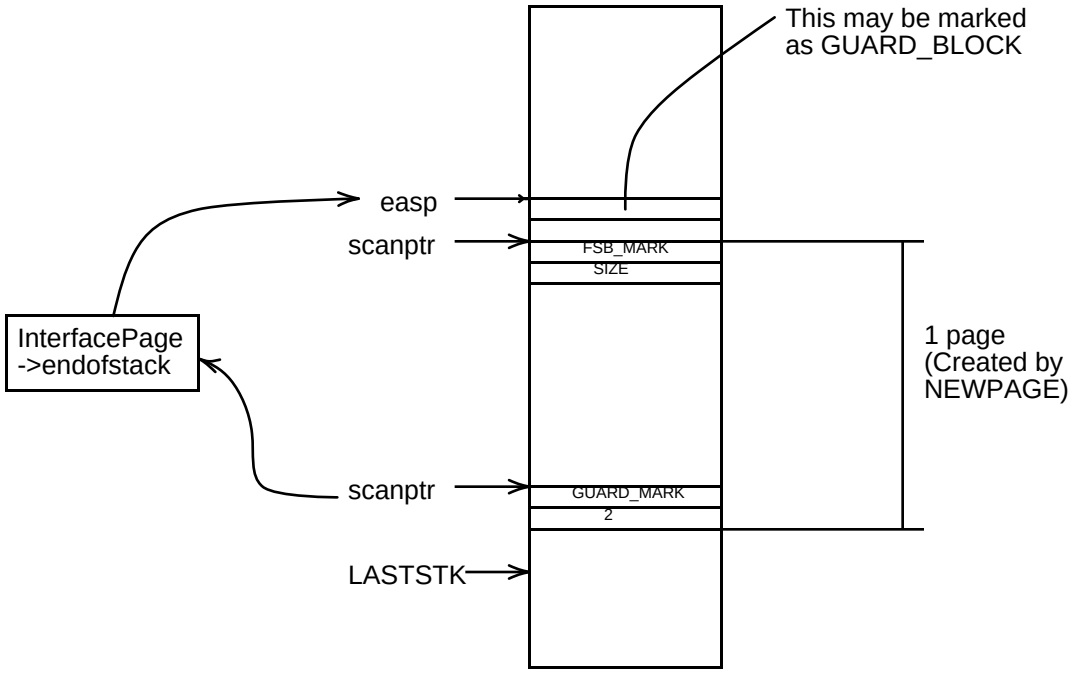
B = { conspage.c , stack.c }

C = { car-cdr.c }

D = { funcall.c }

= READ,PRINT,LOADER


extendstack()
(easp > GUARDSTACKADDR)
&& (! Stackiverflow_flg)
Interruptstate->stackoverflow=T
Stackoverflow_flg=Pendinginterrupt=T
scanptr = easp+2
MAKEFREEBLOCK(scanptr,(DLWORDSPER_PAGE -2))
SETUPGUARDBLOCK((scanptr=easp+DLWORDSPRE_PAGE , 2)
InterfacePage->endofstack = scanptr
MAKEFREEBLOCK(easp,2)
(easp = Interfacepage->endofstack)
< LASTSTACKADDR
return(scanptr)
return(NIL)


This may be marked
as GUARD_BLOCK
easp
scanptr
FSB_MARK
SIZE
InterfacePage
->endofstack
1 page
(Created by
NEWPAGE)
scanptr
GUARD_MARK
2
LASTSTK

FREESTACKBLOCK(n,start,align)

wantedsize = n + STACKAREASIZE + MINEXTRASTACKWORD

easp = InterfacePage->endofstack

start > InterfacePage->stackbase

\StackAreaSize(768)

STARTOVER

start

scanptr= start

scanptr= InterfacePage->stackbase

SCAN

switch (scanptr->flags)

STK_FSB

goto FREESCAN

STK_GUARD

scanptr<easp

goto FREESCAN

scanptr = InterfacePage->stackbase

start

got oSCAN

goto NEWPAGE

STK_FX

scanptr=scanptr->nextblock

break;

default

orig = scanptr

while(scanptr->flags != BF_MARK)

scanptr->flags !=0

error()

scanptr +=DLWORDSPER_CELL

scanptr!=orig

error()

scanptr ->residual

!(bf_checked() && (orig== scanptr->ivar)

error()

scanptr +=DLWORDSPER_CELL

NEXT

((scanptr != start)

(scanptr >= easp)

error()

goto SCAN

NEXTPAGE

easp=extendstack()

goto STARTOVER

error("MP9319)

FREESCAN

freeptr=scanptr

freesize=scanptr->size

FREE

scanptr=freeptr+freesize

switch (scanptr->flag)

STK_FSB

freesize+=scanptr->size

goto FREE

STK_GUARD

scanptr<easp

freesize+=scanptr->size

goto FREE

(align==NIL) || (align==(freeptr%DLWORDSPER_QUAD)

wantedsize=MINEXTRASTACKWORDS

wantedsize=DLWORDSPER_CELL+MINEXTRASTACKWORDS

scanptr=freeptr+wantedsize

SETUPGUARDBLOCK(scanptr,n)

MAKEFREEBLOCK(freeptr,wantedsize)

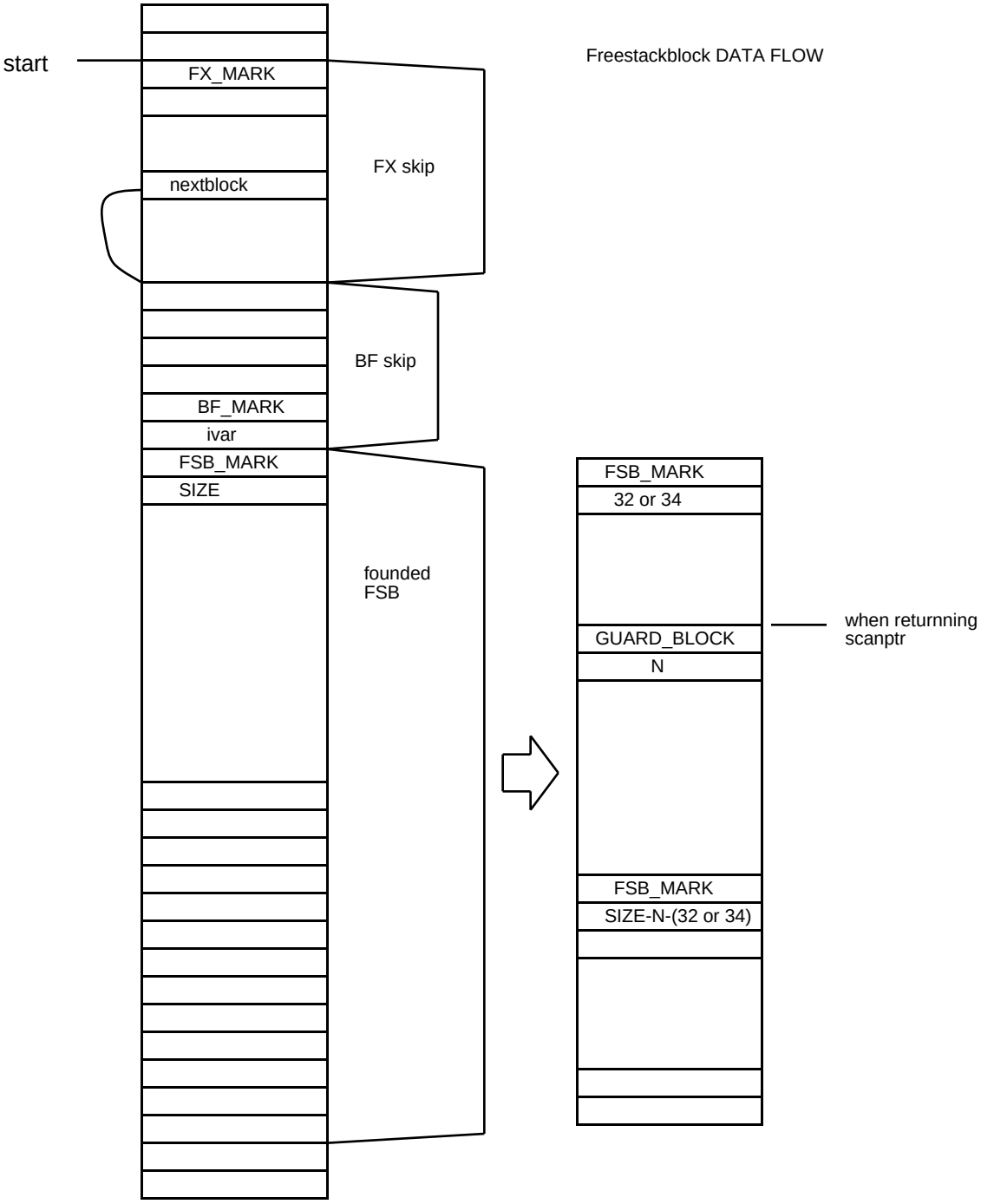
MAKEFREEBLOCK(scanptr+n,,freesize--wantedsize-n)

freesize >= wantedsize

return(scanptr)

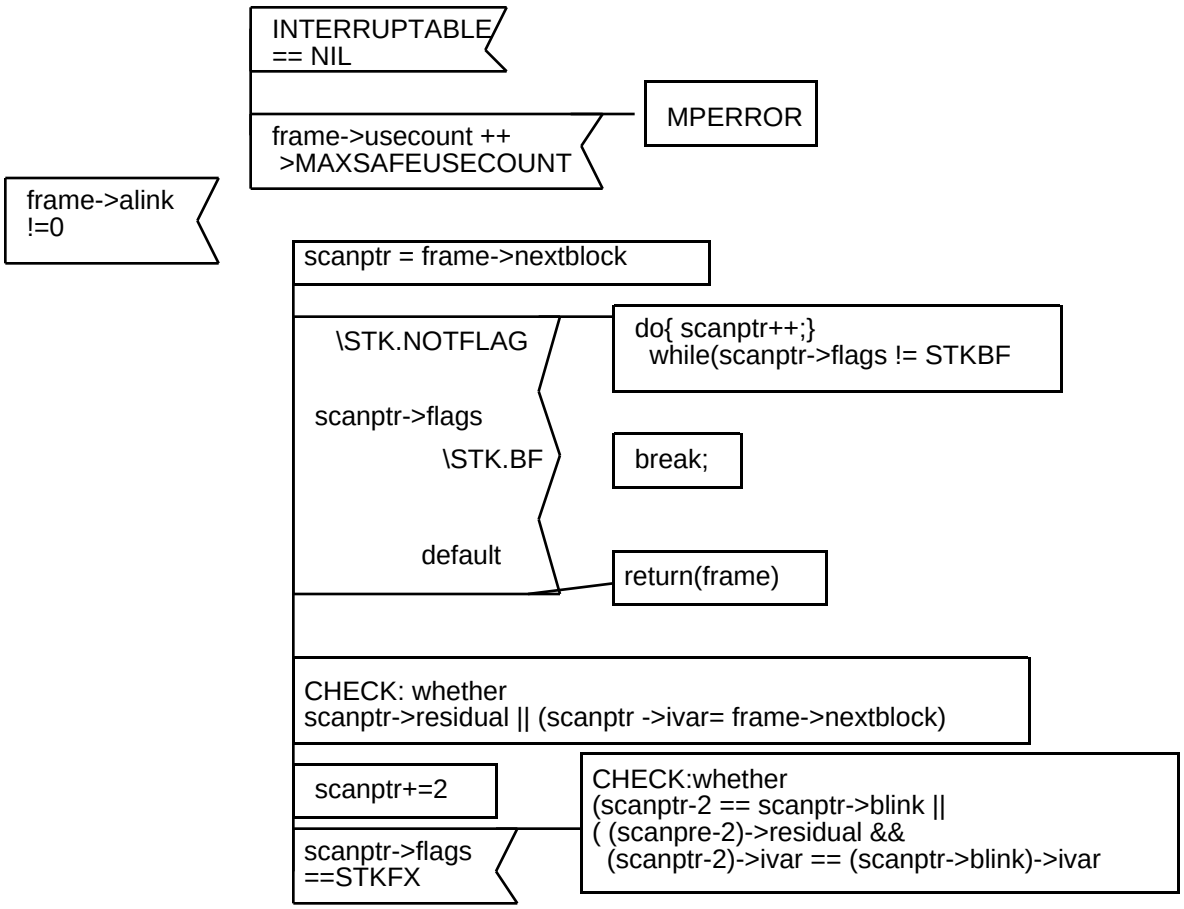
goto NEXT

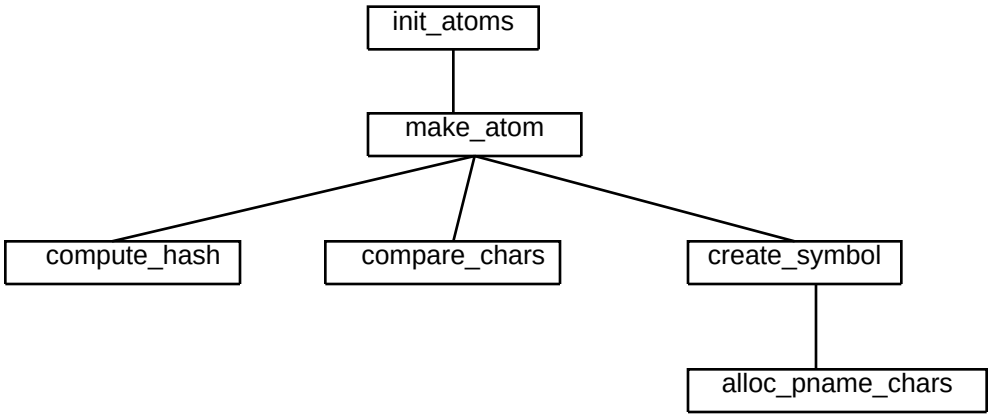
\MinExtraStackWords(32)


start
Freestackblock DATA FLOW
FX_MARK
FX skip
nextblock
BF skip
BF_MARK
ivar
FSB_MARK
SIZE
founded
FSB
FSB_MARK
32 or 34
GUARD_BLOCK
when returnning
scanptr
N
FSB_MARK
SIZE-N-(32 or 34)

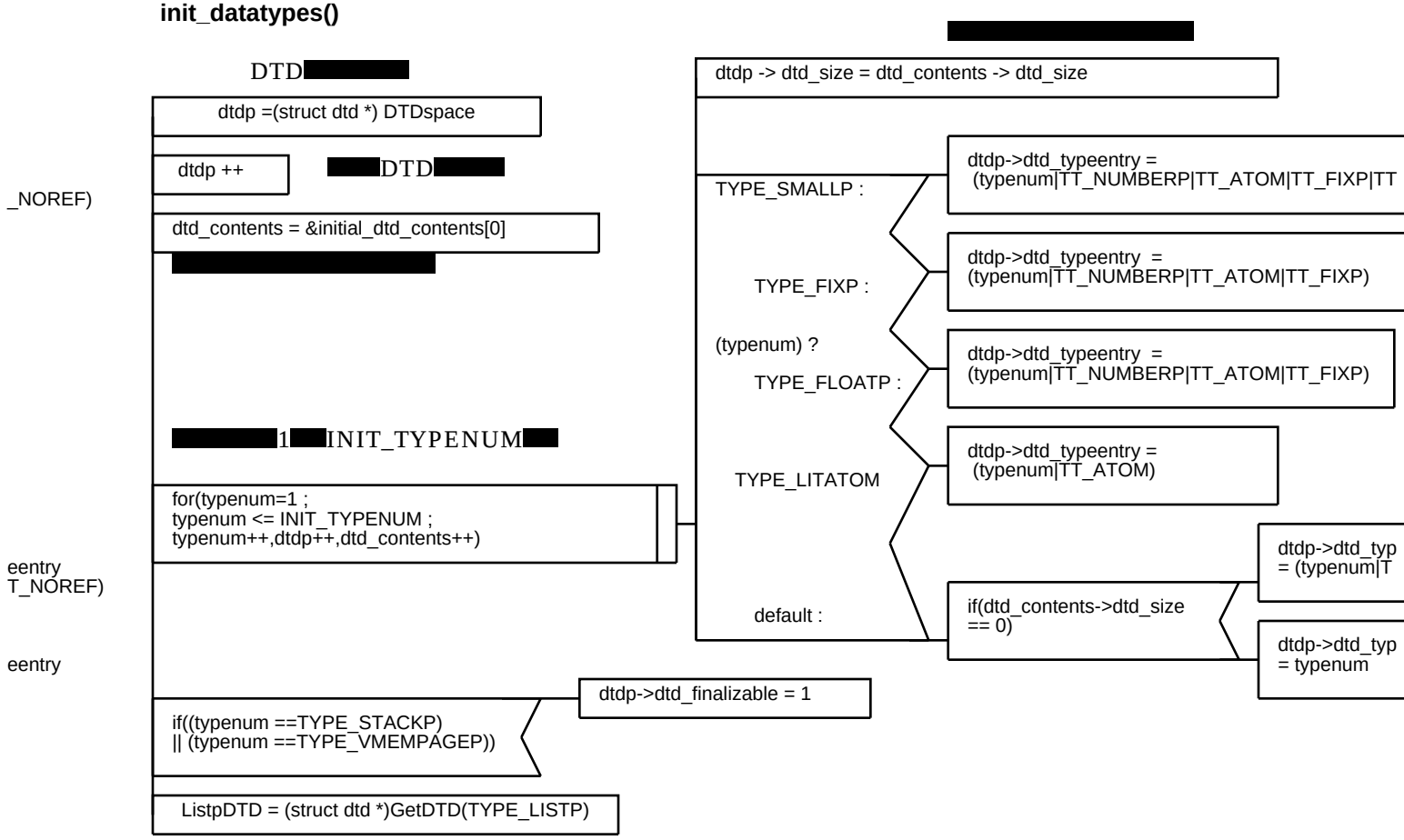
\INCUSECOUNT (FRAME)

struct frameex1 *frame

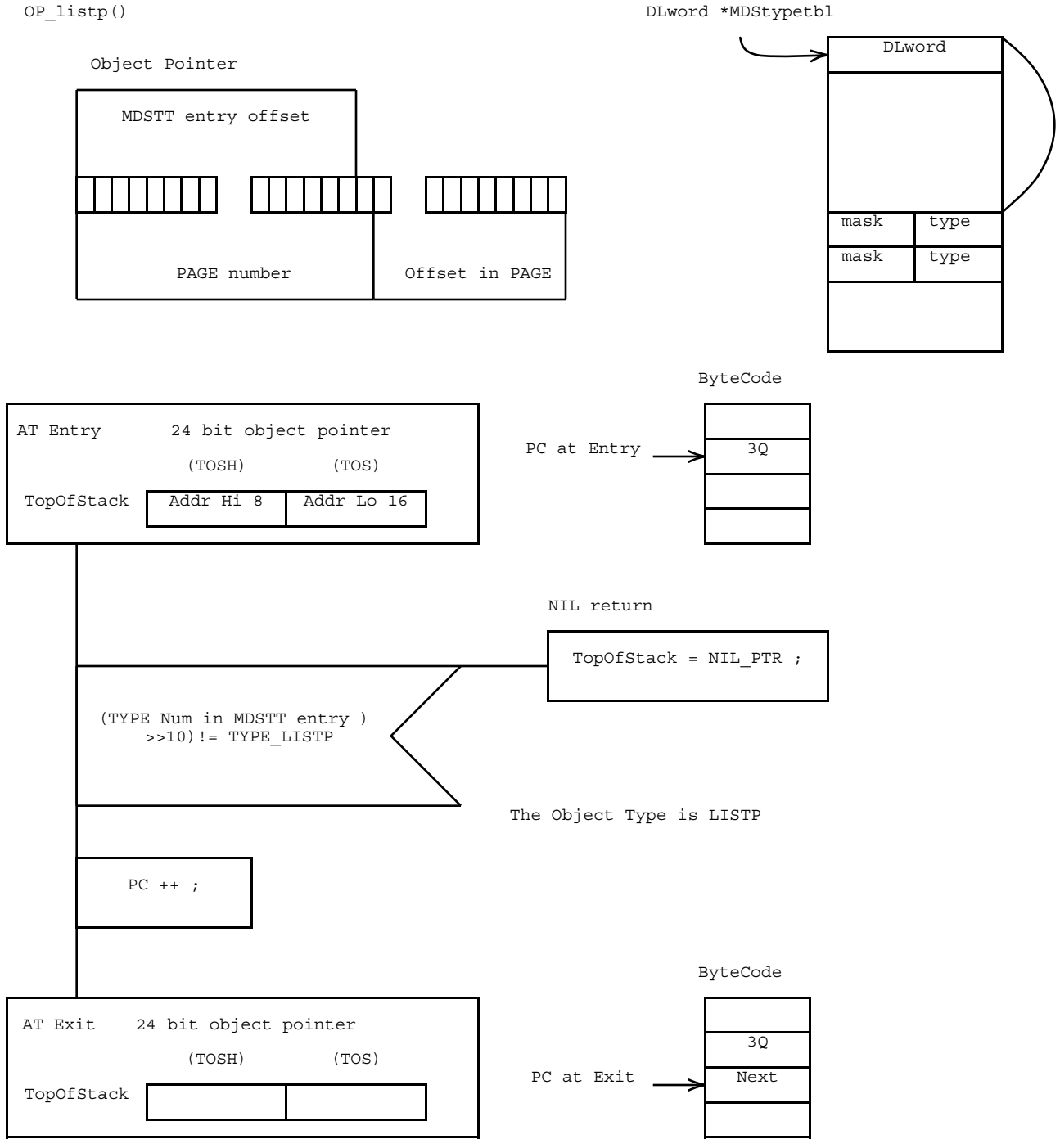


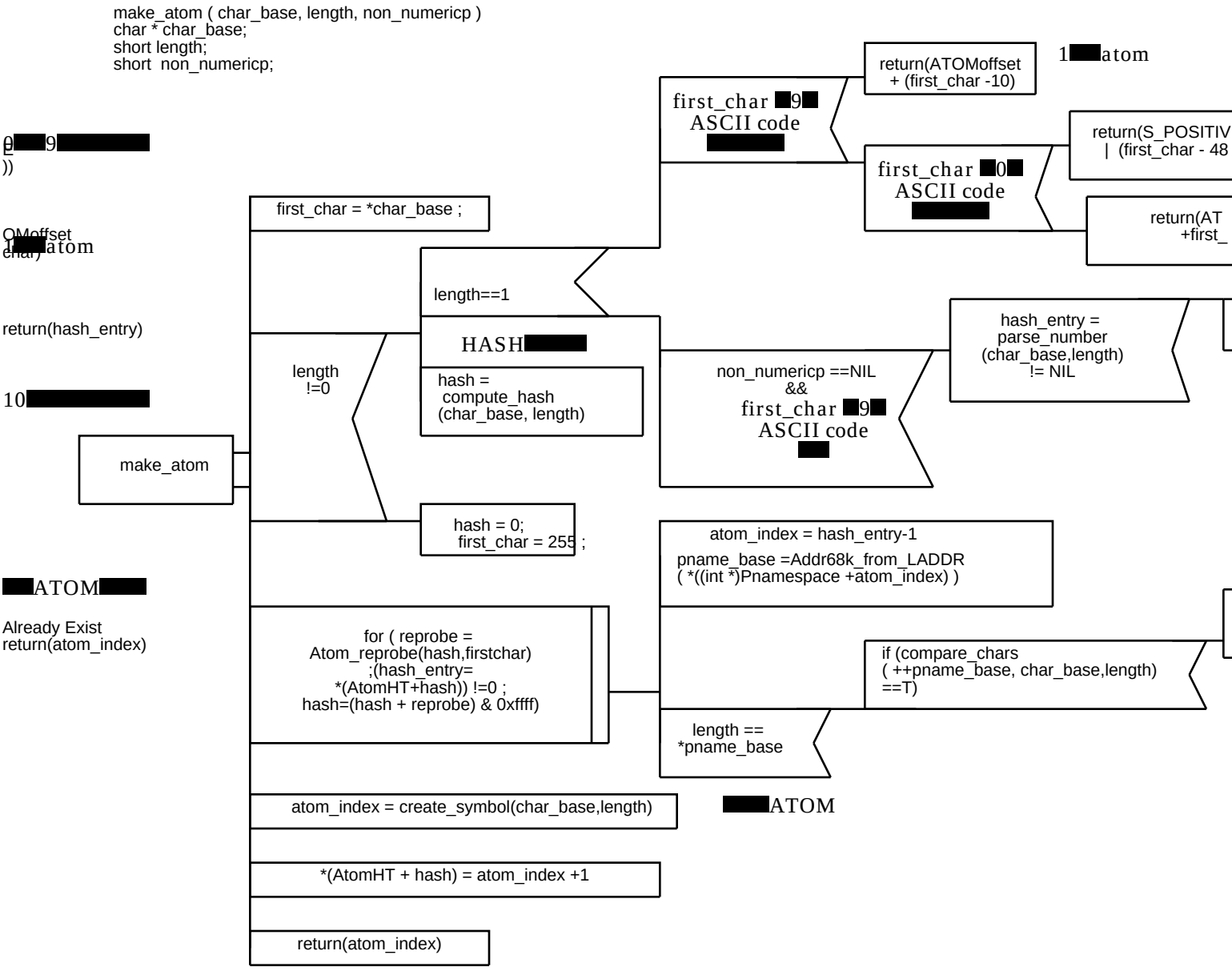

init_atoms
make_atom
compute_hash
compare_chars
create_symbol
alloc_pname_chars

init_datatypes()

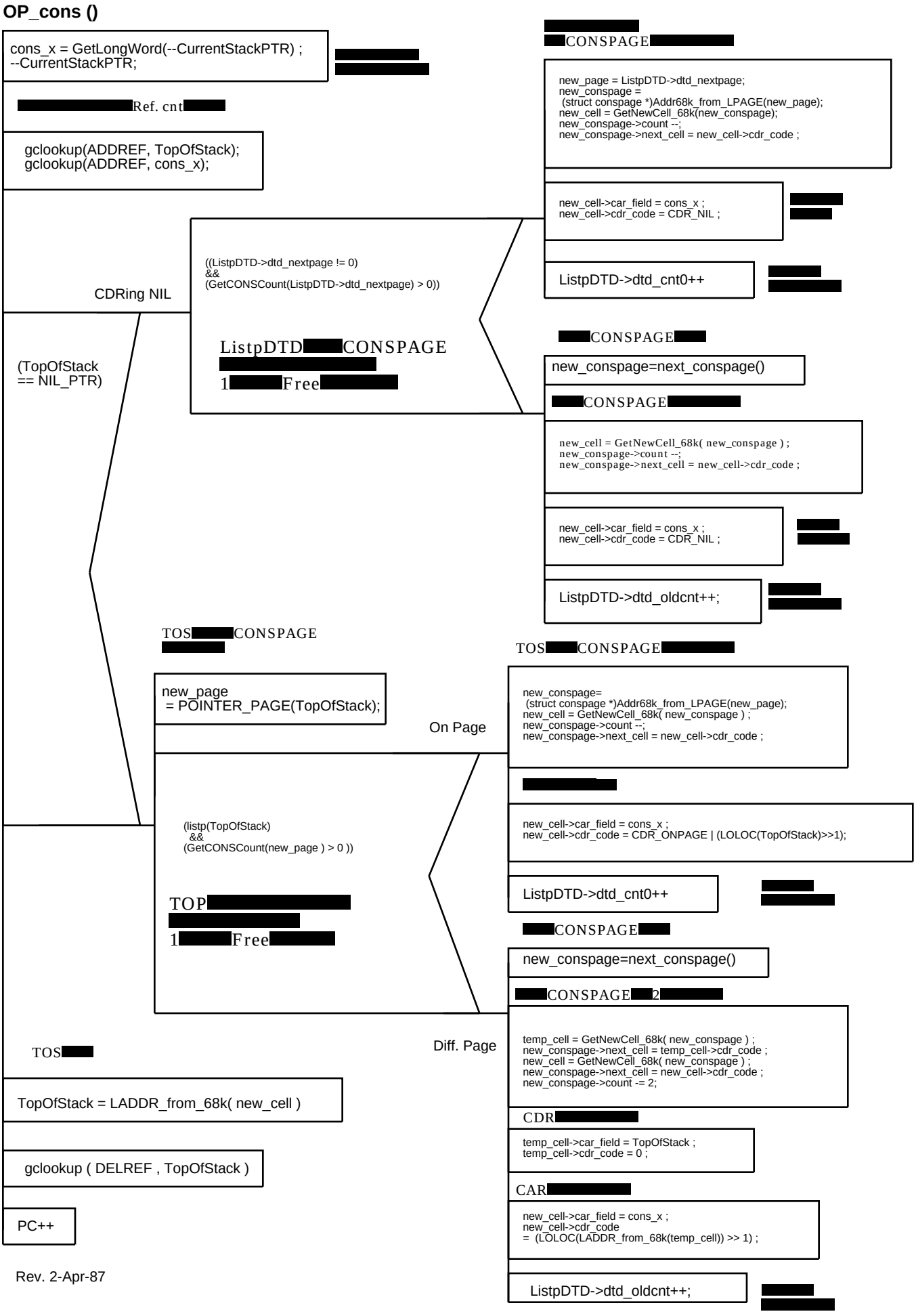


SPEC of LISTP(3Q)



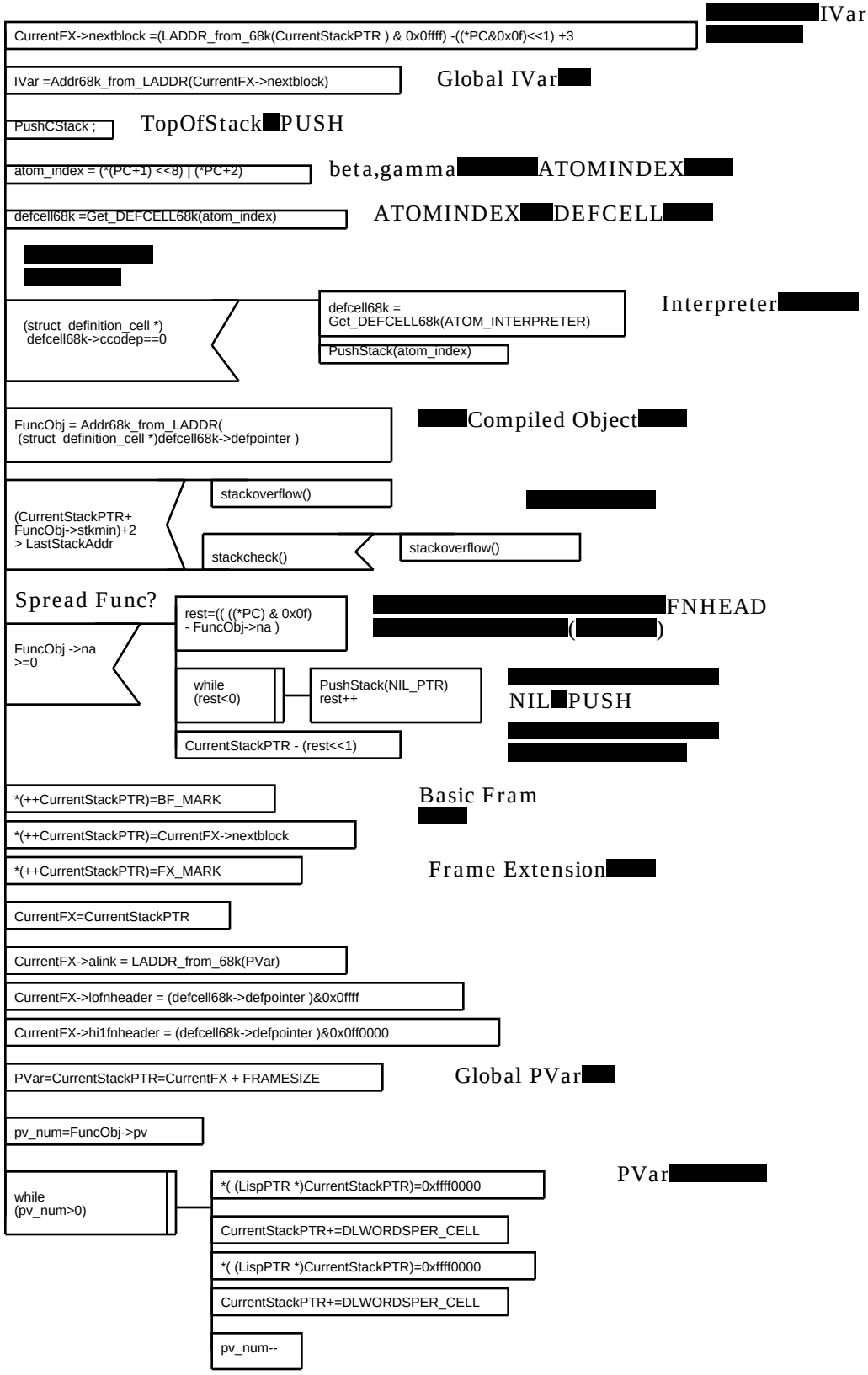

make_atom (char_base, length, non_numericp)
char * char_base;
short length;
short non_numericp;
make_atom
first_char = *char_base ;
length==1
length !=0
HASH
hash = compute_hash (char_base, length)
hash = 0; first_char = 255 ;
non_numericp ==NIL && first_char 9 ASCII code
hash_entry = parse_number (char_base,length) != NIL
first_char 9 ASCII code
return(ATOMoffset + (first_char -10)
first_char 0 ASCII code
return(S_POSITIV | (first_char - 48
return(AT +first_
1 atom
atom_index = hash_entry-1
pname_base =Addr68k_from_LADDR (*((int *)Pnamespace +atom_index))
for (reprobe = Atom_reprobe(hash,firstchar) ;(hash_entry= *(AtomHT+hash)) !=0 ; hash=(hash + reprobe) & 0xffff)
if (compare_chars (++pname_base, char_base,length) ==T)
length == *pname_base
ATOM
atom_index = create_symbol(char_base,length)
*(AtomHT + hash) = atom_index +1
return(atom_index)
ATOM
Already Exist
return(atom_index)
return(hash_entry)

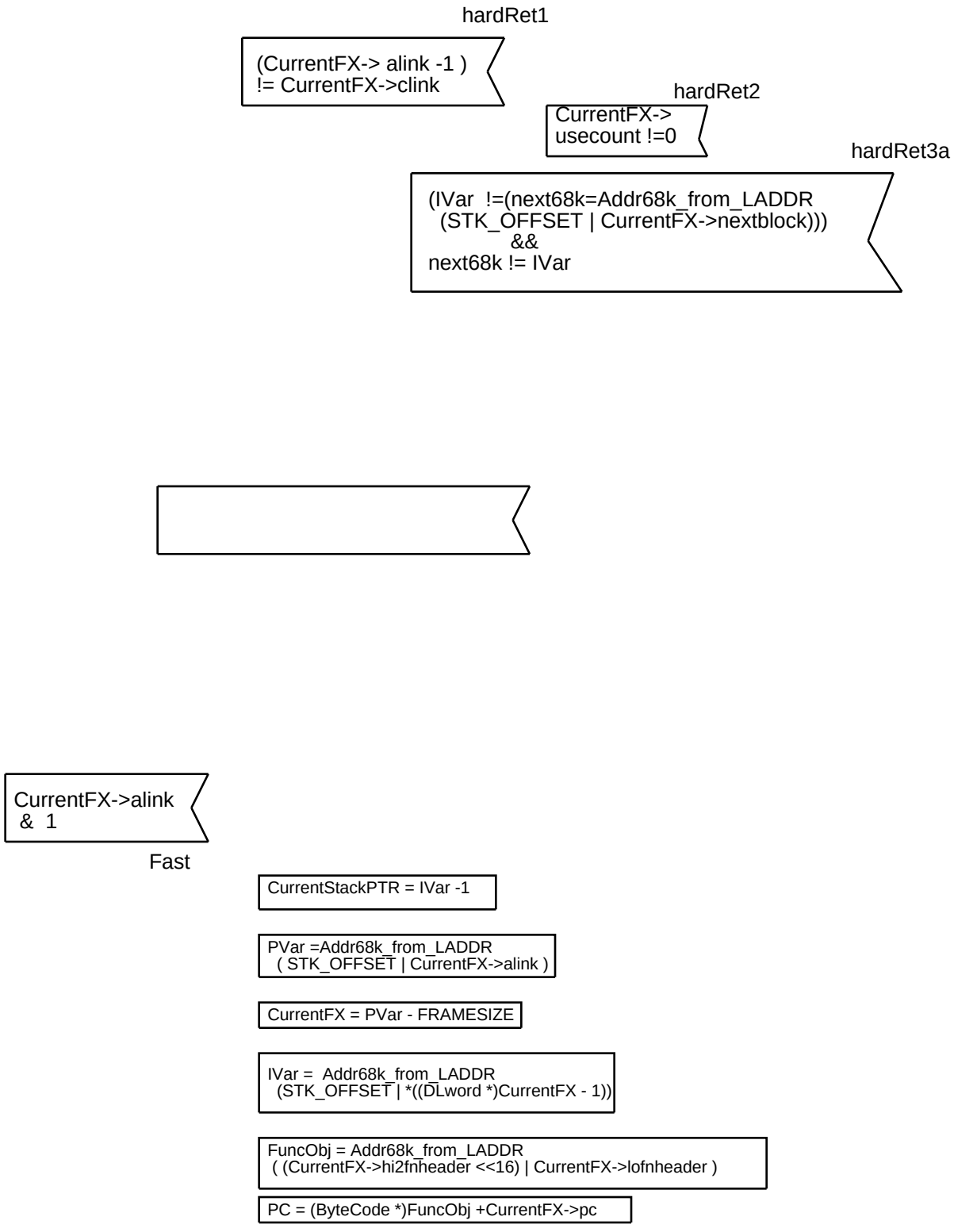
OP_cons ()



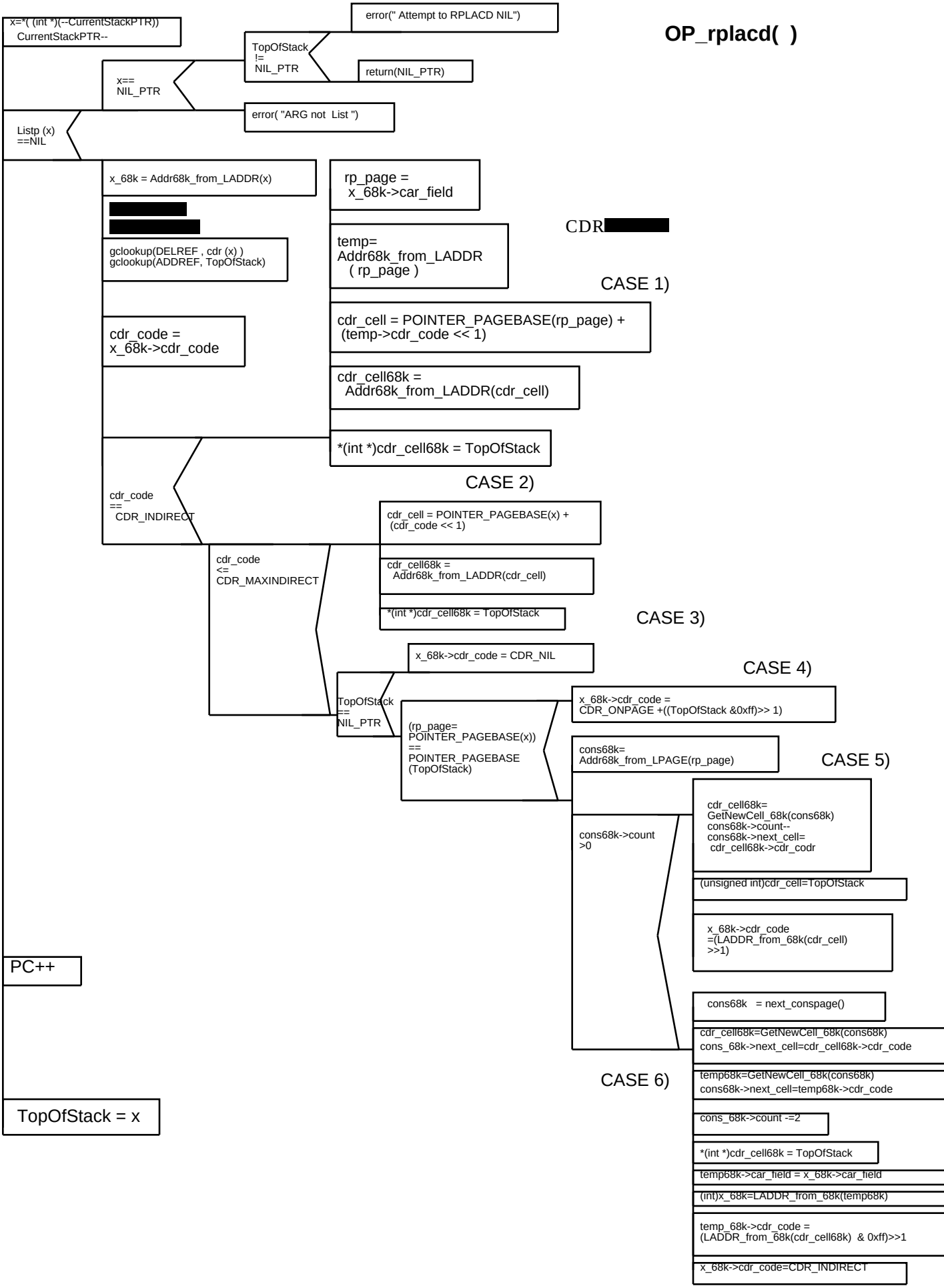
Rev. 2-Apr-87

OP_fn


CurrentFX->nextblock =(LADDR_from_68k(CurrentStackPTR) & 0x0ffff) -((*PC&0x0f)<<1) +3
IVar
IVar =Addr68k_from_LADDR(CurrentFX->nextblock)
Global IVar
PushCStack ;
TopOfStack PUSH
atom_index = (*(PC+1) <<8) | (*PC+2)
beta,gamma ATOMINDEX
defcell68k =Get_DEFCELL68k(atom_index)
ATOMINDEX DEFCELL
(struct definition_cell *)
defcell68k->ccodep==0
defcell68k =
Get_DEFCELL68k(ATOM_INTERPRETER)
PushStack(atom_index)
Interpreter
FuncObj = Addr68k_from_LADDR(
(struct definition_cell *)defcell68k->defpointer)
Compiled Object
(CurrentStackPTR+
FuncObj->stkmin)+2
> LastStackAddr
stackoverflow()
stackcheck()
stackoverflow()
Spread Func?
rest=((((*PC) & 0x0f)
- FuncObj->na)
FNHEAD
FuncObj ->na
>=0
while
(rest<0)
PushStack(NIL_PTR)
rest++
NIL PUSH
CurrentStackPTR - (rest<<1)
*(++CurrentStackPTR)=BF_MARK
Basic Fram
*(++CurrentStackPTR)=CurrentFX->nextblock
*(++CurrentStackPTR)=FX_MARK
Frame Extension
CurrentFX=CurrentStackPTR
CurrentFX->alink = LADDR_from_68k(PVar)
CurrentFX->lofnheader = (defcell68k->defpointer)&0x0ffff
CurrentFX->hi1fnheader = (defcell68k->defpointer)&0x0ff0000
PVar=CurrentStackPTR=CurrentFX + FRAMESIZE
Global PVar
pv_num=FuncObj->pv
PVar
while
(pv_num>0)
*((LispPTR *)CurrentStackPTR)=0xffff0000
CurrentStackPTR+=DLWORDSPER_CELL
*((LispPTR *)CurrentStackPTR)=0xffff0000
CurrentStackPTR+=DLWORDSPER_CELL
pv_num--

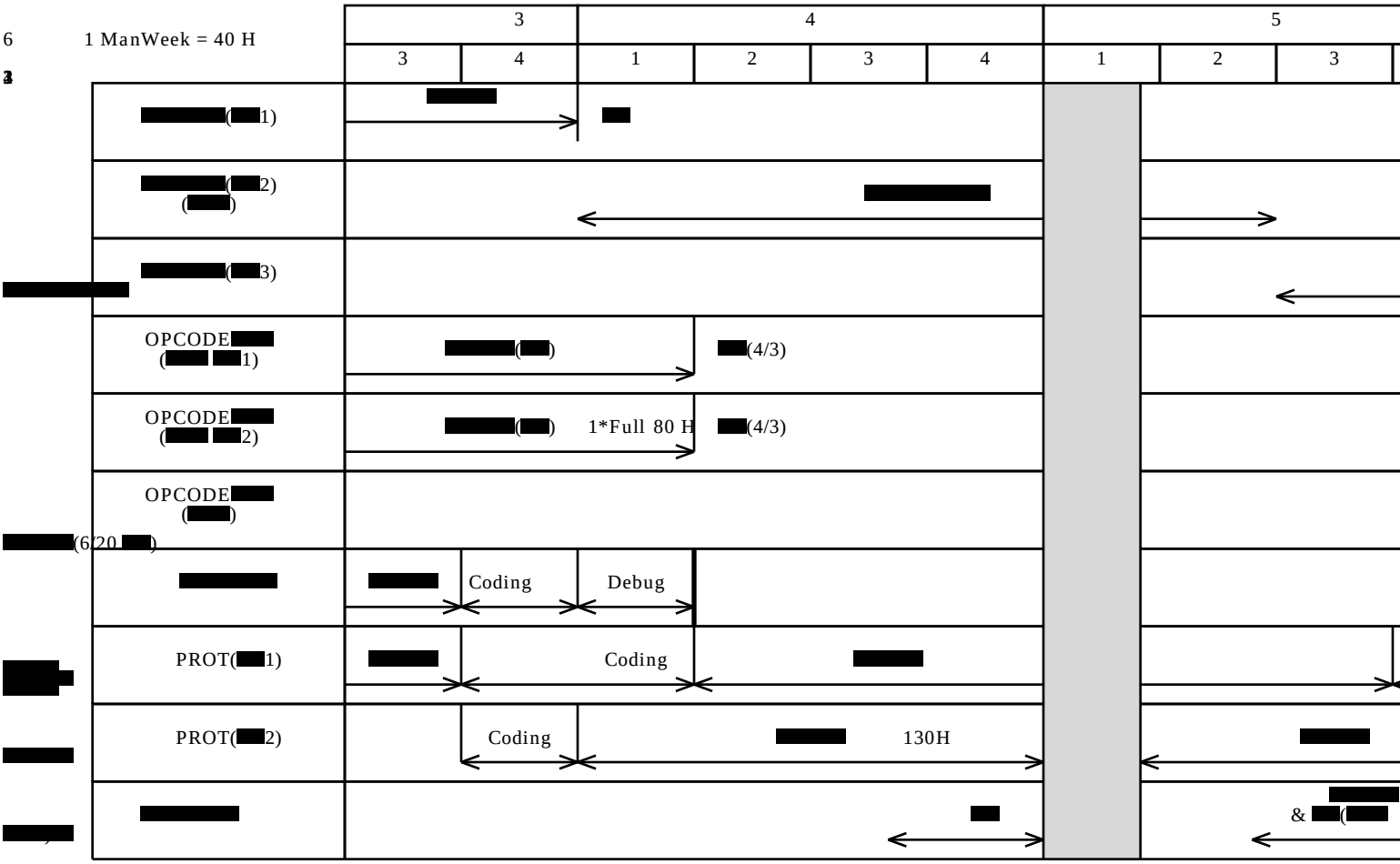

hardRet1
(CurrentFX-> alink -1)
!= CurrentFX->clink
hardRet2
CurrentFX->
usecount !=0
hardRet3a
(IVar !=(next68k=Addr68k_from_LADDR
(STK_OFFSET | CurrentFX->nextblock)))
&&
next68k != IVar
CurrentFX->alink
& 1
Fast
CurrentStackPTR = IVar -1
PVar =Addr68k_from_LADDR
(STK_OFFSET | CurrentFX->alink)
CurrentFX = 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

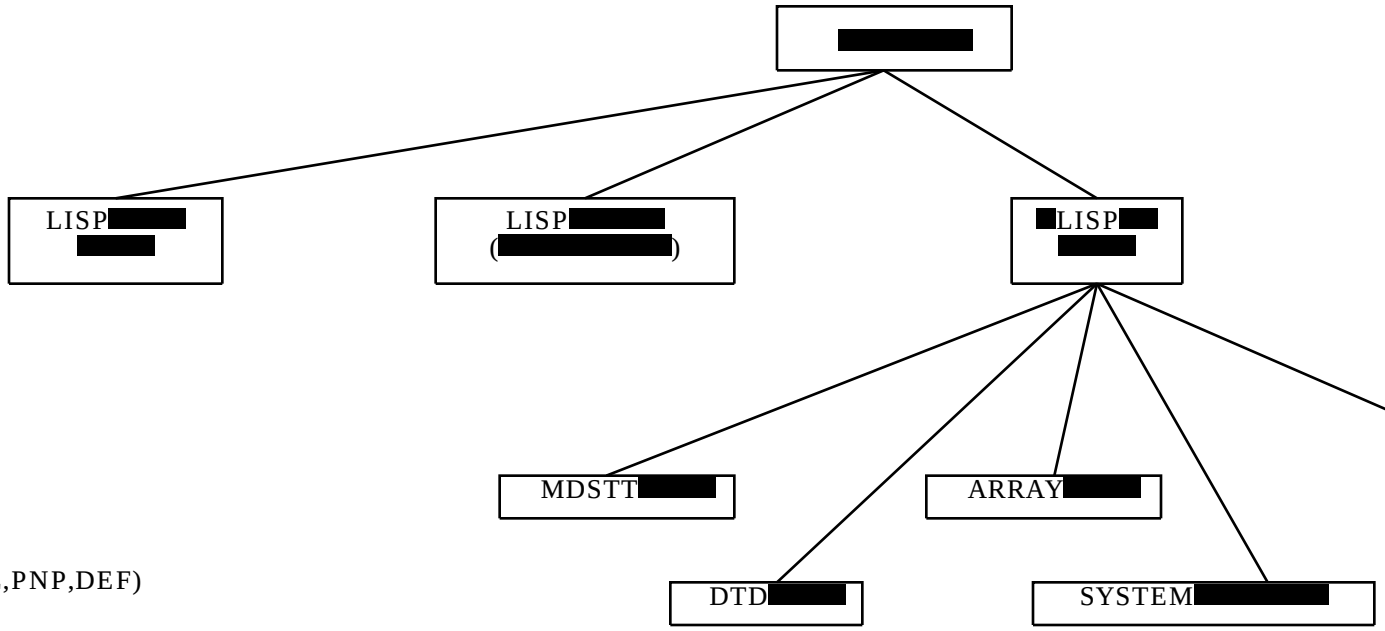
OP_rplacd( )


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)
rp_page =
x_68k->car_field
CDR
gclookup(DELREF , cdr (x))
gclookup(ADDREF, TopOfStack)
temp=
Addr68k_from_LADDR
(rp_page)
CASE 1)
cdr_code =
x_68k->cdr_code
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
(rp_page=
POINTER_PAGEBASE(x))
==
POINTER_PAGEBASE
(TopOfStack)
x_68k->cdr_code =
CDR_ONPAGE +((TopOfStack &0xff)>> 1)
cons68k=
Addr68k_from_LPAGE(rp_page)
CASE 5)
cons68k->count
>0
cdr_cell68k=
GetNewCell_68k(cons68k)
cons68k->count--
cons68k->next_cell=
cdr_cell68k->cdr_codr
(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
temp68k=GetNewCell_68k(cons68k)
cons68k->next_cell=temp68k->cdr_code
CASE 6)
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

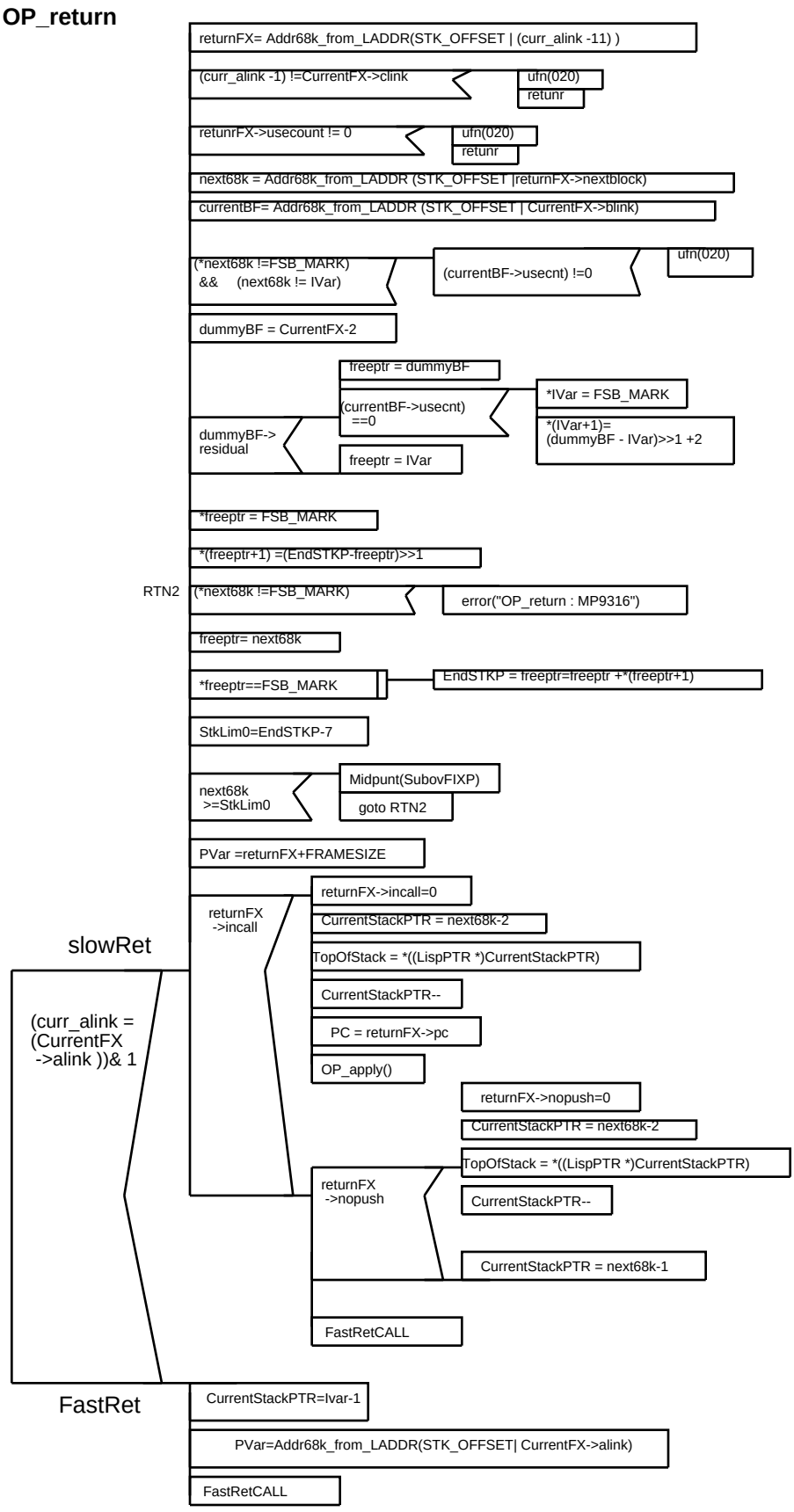
Revised 24 Mar. 1987

T.Shimizu

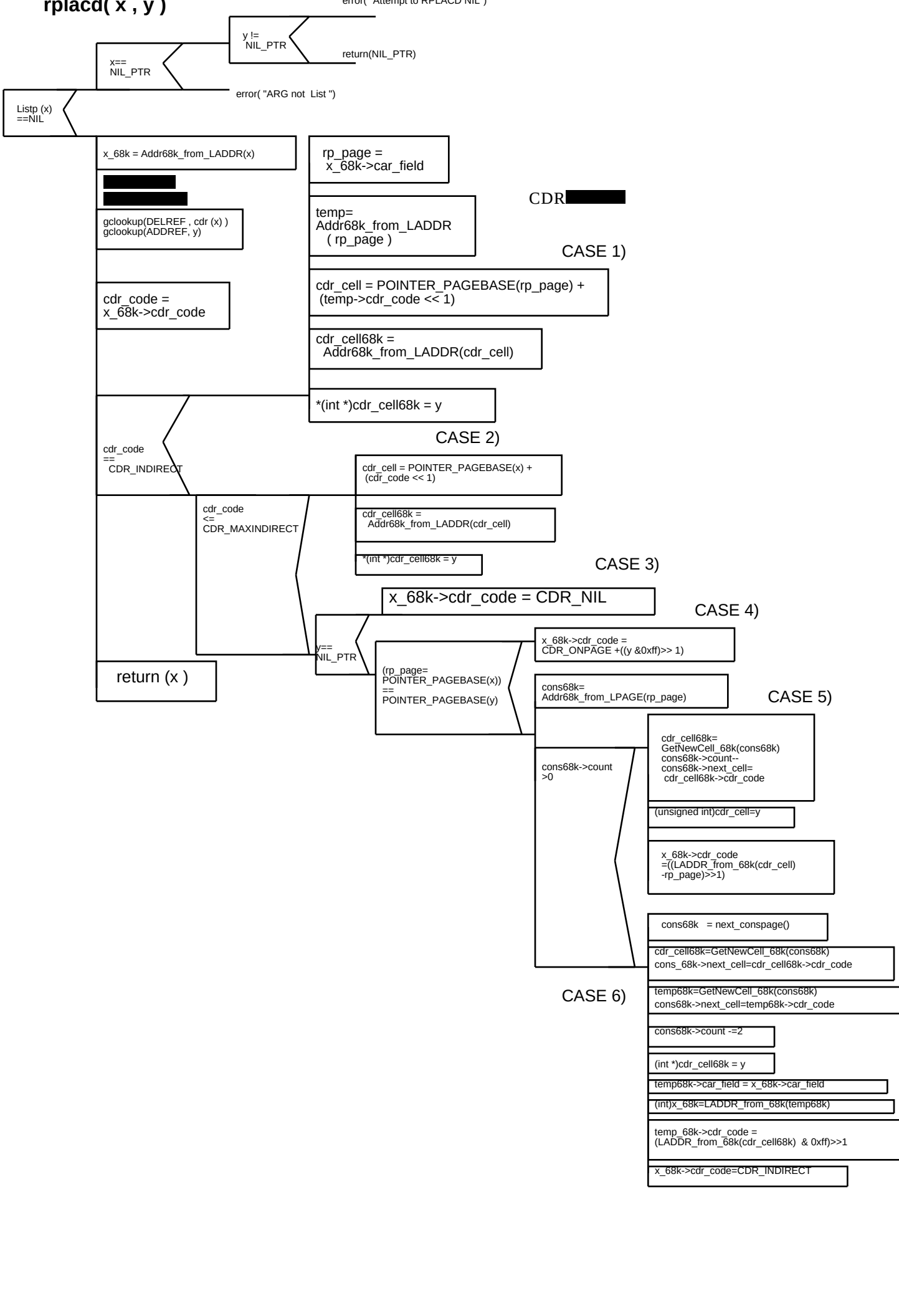


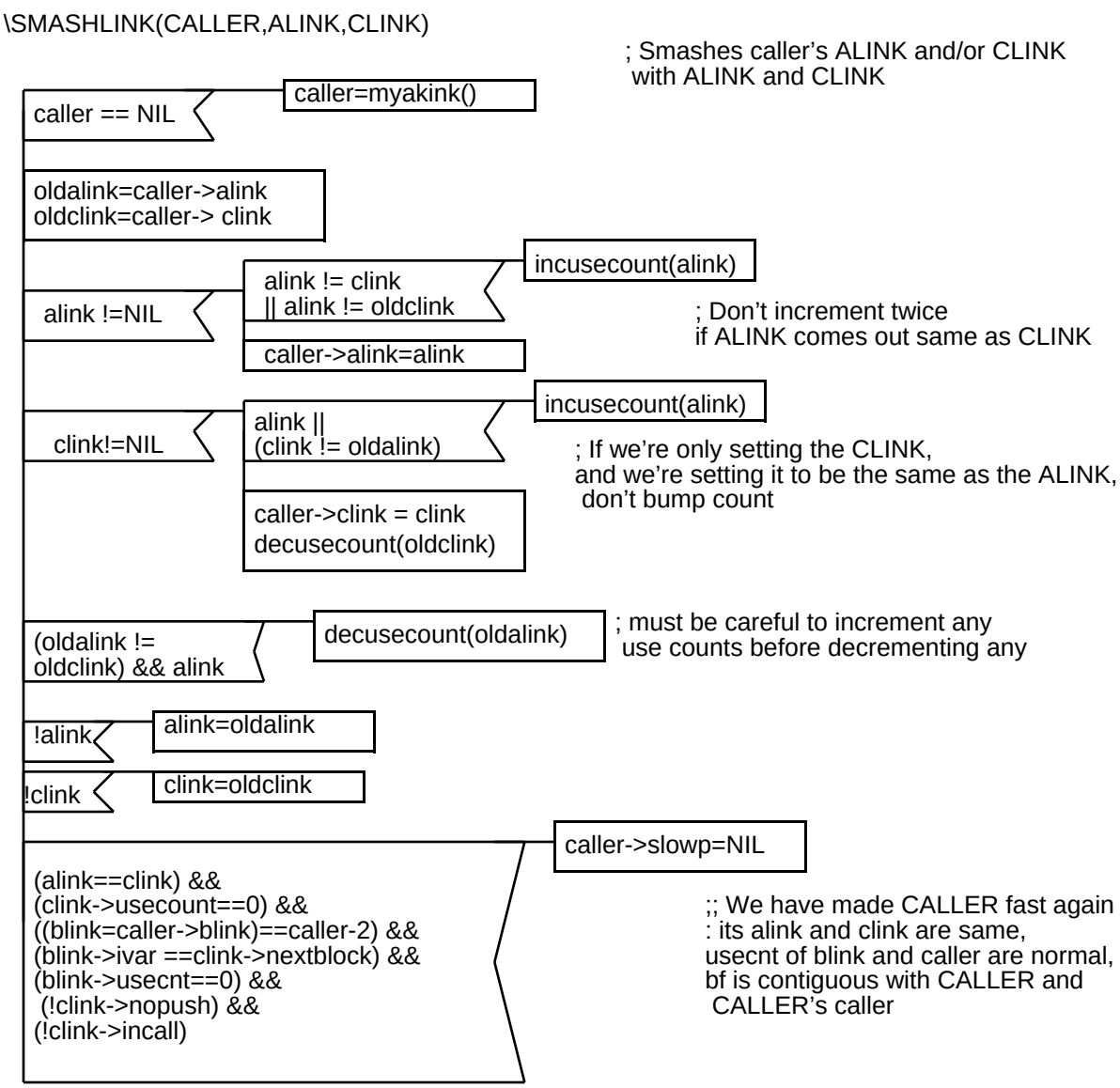

LISP
LISP
(PLIST,VAL,PNP,DEF)

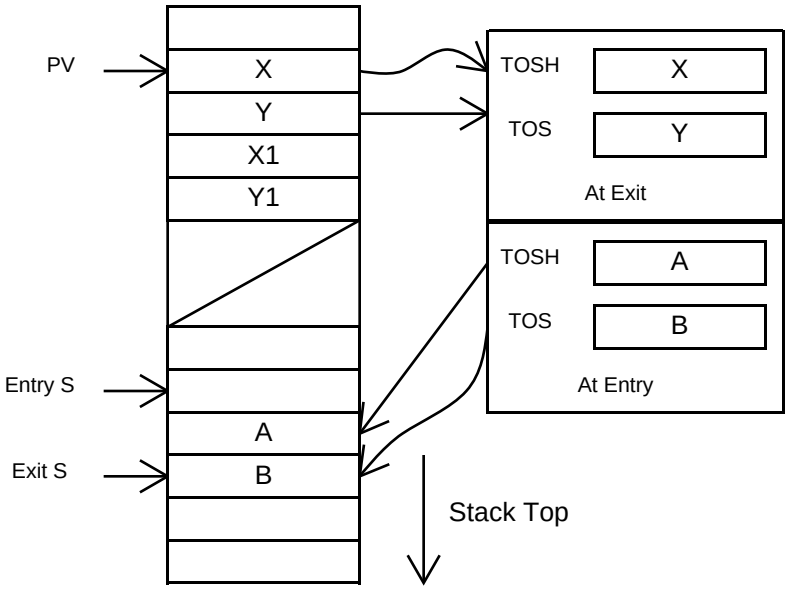
OP_return

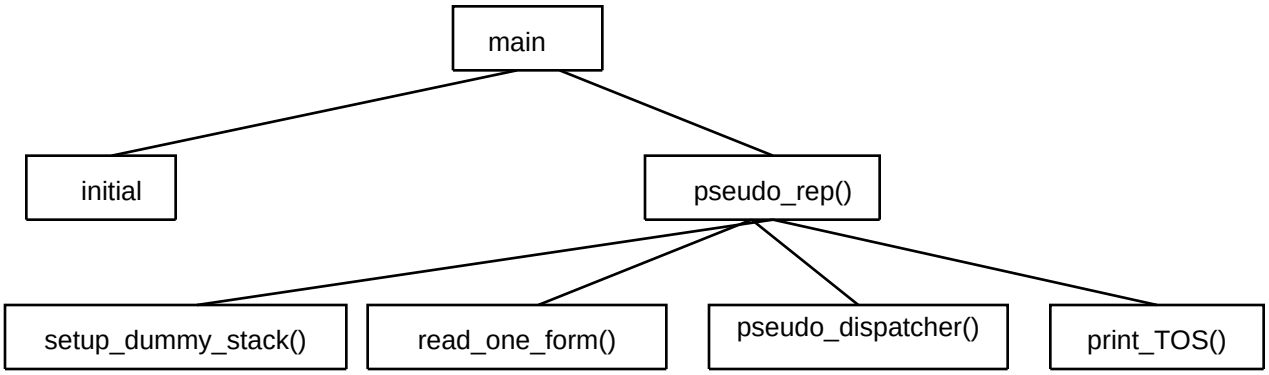

returnFX= Addr68k_from_LADDR(STK_OFFSET | (curr_alink -11))
(curr_alink -1) !=CurrentFX->clink
ufn(020)
retunr
retunrFX->usecount != 0
ufn(020)
retunr
next68k = Addr68k_from_LADDR (STK_OFFSET |returnFX->nextblock)
currentBF= Addr68k_from_LADDR (STK_OFFSET | CurrentFX->blink)
(*next68k !=FSB_MARK)
&& (next68k != IVar)
(currentBF->usecnt) !=0
ufn(020)
dummyBF = CurrentFX-2
freeptr = dummyBF
*IVar = FSB_MARK
(currentBF->usecnt)
==0
dummyBF->
residual
*(IVar+1)=
(dummyBF - IVar)>>1 +2
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
returnFX->incall=0
CurrentStackPTR = next68k-2
TopOfStack = *((LispPTR *)CurrentStackPTR)
CurrentStackPTR--
PC = returnFX->pc
OP_apply()
slowRet
(curr_alink =
(CurrentFX
->alink))& 1
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)
rp_page = x_68k->car_field
CDR
gclookup(DELREF , cdr (x))
gclookup(ADDREF, y)
temp= Addr68k_from_LADDR (rp_page)
CASE 1)
cdr_code = x_68k->cdr_code
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_code <= CDR_MAXINDIRECT
cdr_cell68k = Addr68k_from_LADDR(cdr_cell)
*(int *)cdr_cell68k = y
CASE 3)
x_68k->cdr_code = CDR_NIL
CASE 4)
y== NIL_PTR
x_68k->cdr_code = CDR_ONPAGE +((y &0xff)>> 1)
(rp_page= POINTER_PAGEBASE(x)) == POINTER_PAGEBASE(y)
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)
; Don't increment twice
if ALINK comes out same as CLINK
caller->alink=alink
clink!=NIL
alink ||
(clink != oldalink)
incusecount(alink)
; If we're only setting the CLINK,
and we're setting it to be the same as the ALINK,
don't bump count
caller->clink = clink
decusecount(oldclink)
(oldalink !=
oldclink) && alink
decusecount(oldalink)
; must be careful to increment any
use counts before decrementing any
!alink
alink=oldalink
!clink
clink=oldclink
caller->slowp=NIL
(alink==clink) &&
(clink->usecount==0) &&
((blink=caller->blink)==caller-2) &&
(blink->ivar ==clink->nextblock) &&
(blink->usecnt==0) &&
(!clink->nopush) &&
(!clink->incall)
;; 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
X
Y
X1
Y1
Entry S
A
Exit S
B
Stack Top
TOSH
X
TOS
Y
At Exit
TOSH
A
TOS
B
At Entry


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

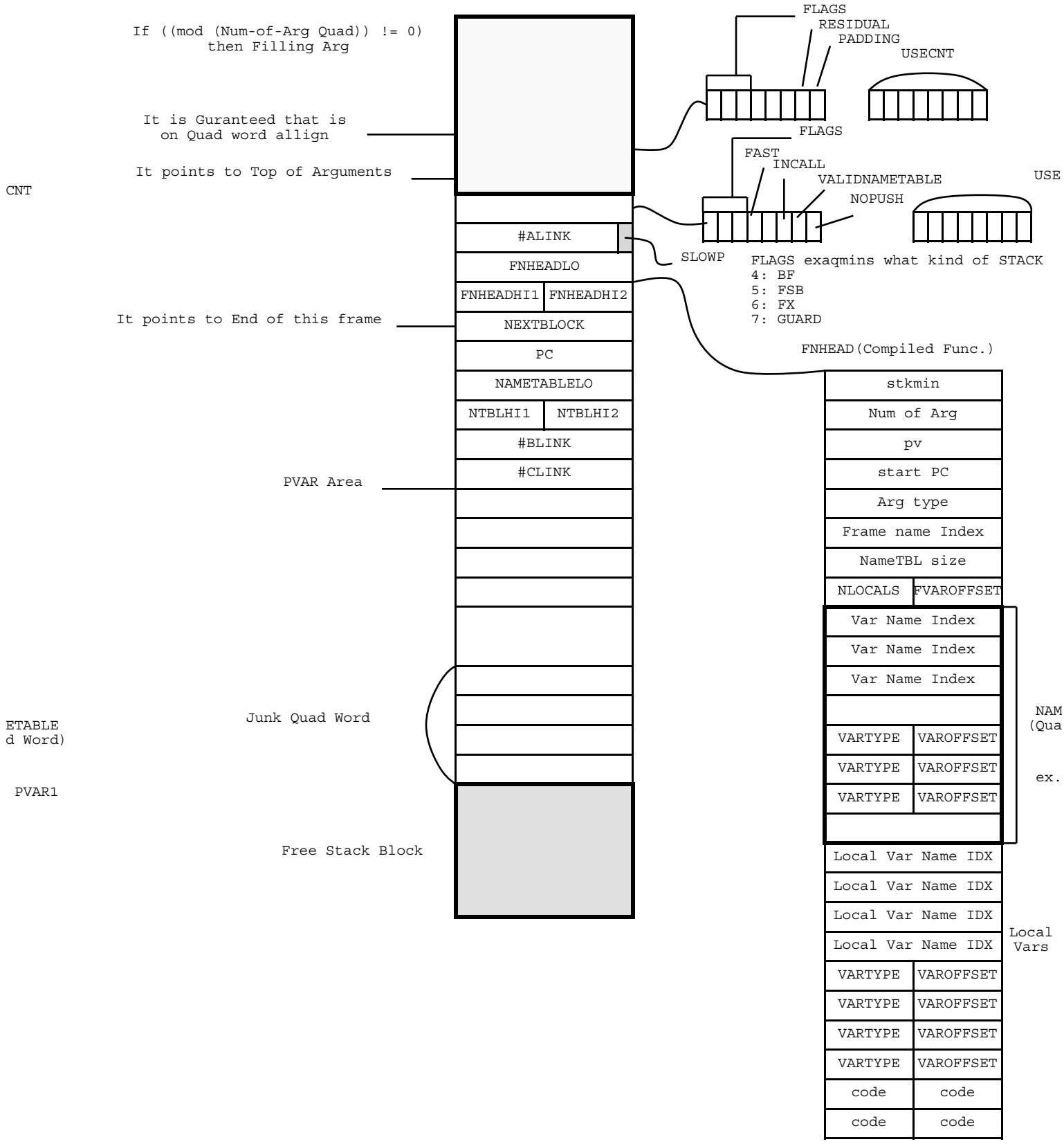
NSFILING is definitely proprietary (implementation of Filing protocol).

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

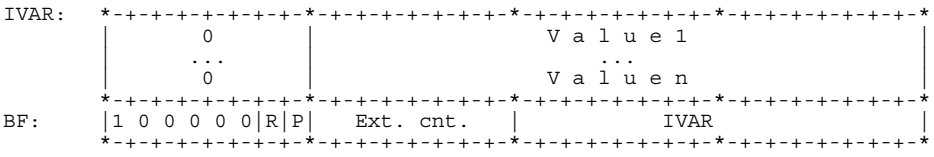
original: August 15, 1981 by Bill van Melle, Larry Masinter
reformatted for implementor's manual: July 2, 1985 by Ron Fischer
revised: August 1, 1985 by Bill van Melle
revised: July 16, 1988 by Frank Shih (minor note re: Maiko)

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.



The fields are interpreted as follows:

IVAR — The 2nd word of the BF cell points back at the IVAR.

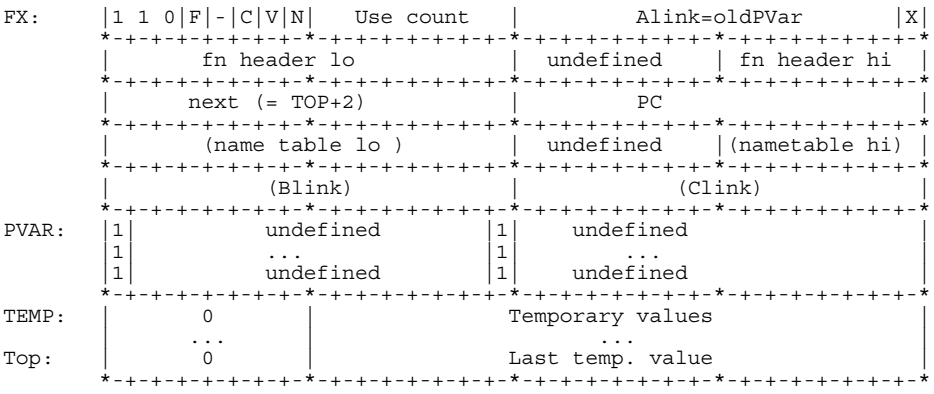
R — "Residual". This bit is on in "dummy" basic frames which are actually only 2-word quantities. Normal function call leaves the bit off. Dummy BF's are created when an FX has to move or be copied (on stack overflow or returning to a frame with non-zero use count). It's not clear these are really necessary—one 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").



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.

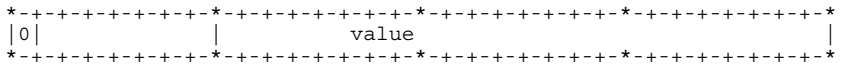
PVAR — This area contains PROG/local variable values and free variable binding pointers. Initialized with all ones in the left half, which denotes {variable not bound} for PROG variables, and {variable not looked up} for free variables (see below). PVAR is quad-aligned (hence so are FX+1 and BF), and 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.

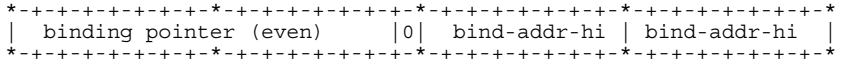
When a function is called, its arguments appear at the end of the caller's FX. The implementation is designed so that the bookkeeping for a BF appears at its high end, and for a FX at its low end, so that the arguments can be made into a new BF without having to copy them. After thus fabricating a new 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:

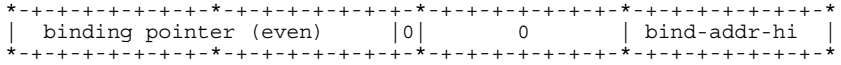


When a free variable is found during free variable lookup, the variable's binding slot is filled in with:



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.:



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.

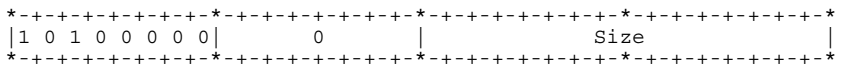
FVAR_ (the free variable assignment opcode), tests whether the binding address of the variable is in stack space. If so, it can do the assignment directly. If not, it must do the assignment using RPLPTR, i.e., decrement the reference count of the old value and increment the reference count of the new. 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.

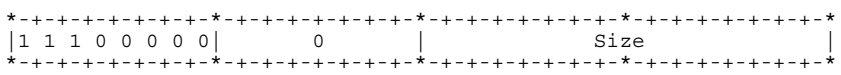
Free blocks

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



Guard block



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:

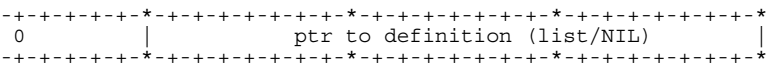
000b	Ordinary pointer (not a flag word)
100b	Basic frame
101b	Free block
110b	Frame extension
111b	Guard block

Function format

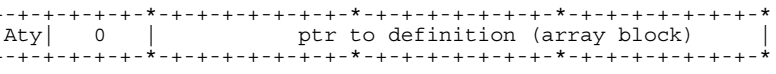
1. Function definition cells

Function definition cells are either:

Interpreted:



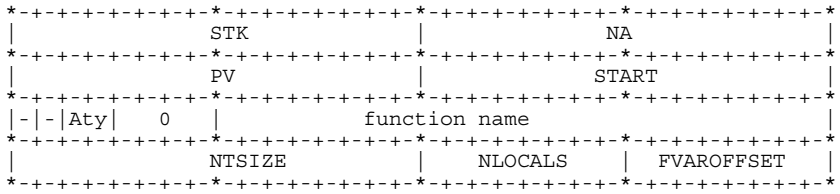
Compiled:



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:



`NA` The number of arguments this function expects. For `LAMBDA*` functions (arbitrary number of args, no adjustment) this field is –1.

`PV` Number of quadwords required for the `PVAR` region, less 1, i.e., (#PVars + #Fvars + 1)/2 – 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.

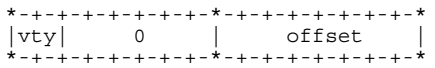
NLOCALS Number of PROG variables.

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 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 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), and then revalidates all of the device's open files. 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 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* = –1 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 CBUFFPTR 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 RESETSAVEs 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* = –1 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' 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 RESETSAVEs 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—ncluding 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 followingstandard 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 X ... X*) [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 VALUE ... VAR VALUE*) [Macro]

Does a SETQ in parallel of *VAR* (unevaluated) to *VALUE* , *VAR* to *VALUE* , etc. All of the *VALUE* 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 VALUE NAME VALUE ...*)

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
- $M \subseteq \square [w] \Leftrightarrow \forall$ v with Rwv: $M \subseteq [v]$

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 ... CLAUSE DEFAULT*) [Function]

Lets you branch one of several ways, based on the character code *E*. The first item in each *CLAUSE* 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 X ... X*) [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 HARRAY ... HARRAY*) [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* , *HARRAY* , ... *HARRAY* . 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 X ... X*) [NoSpread Function]

X + *X* + ... + *X*.

(**MINUS** *X*) [Function]

- *X*

(**DIFFERENCE** *X Y*) [Function]

X - *Y*

(**TIMES** *X X ... X*) [NoSpread Function]

X * *X* * ... * *X*

(**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 X ... X*) [NoSpread Function]

Returns the minimum of *X* , *X* , . . . , *X* . (MIN) returns the value of MAX.INTEGER (see the Integer Arithmetic section below).

(**MAX** *X X ... X*) [NoSpread Function]

Returns the maximum of *X* , *X* , ..., *X* . (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 X ... X*) [NoSpread Function]

Returns the sum *X* + *X* + ... + *X* `(IPLUS) = 0`.

(**IMINUS** *X*) [Function]

-*X*

(**IDIFFERENCE** *X Y*) [Function]

`X - Y`

(**ADD1** *X*) [Function]

`X + 1`

(**SUB1** *X*) [Function]

`X - 1`

(**ITIMES** *X X ... X*) [NoSpread Function]

Returns the product `X * X * ... * X`. `(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 X ... X*) [NoSpread Function]

Returns the minimum of *X*, *X*, ..., *X*. `(IMIN)` returns the largest possible large integer, the value of `MAX.INTEGER`.

(**IMAX** *X X ... X*) [NoSpread Function]

Returns the maximum of *X* , *X* , ..., *X* . (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 N*) [Function]

Returns the greatest common divisor of *N* and *N* , (GCD 72 64)=8.

Logical Arithmetic Functions

(**LOGAND** *X X ... X*) [NoSpread Function]

Returns the logical AND of all its arguments, as an integer. Example:

```
(LOGAND 7 5 6)  =>  4
```

(**LOGOR** *X X ... X*) [NoSpread Function]

Returns the logical OR of all its arguments, as an integer. Example:

```
(LOGOR 1 3 9)  =>  11
```

(**LOGXOR** *X X ... X*) [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; RSHing 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 X ... X*) [NoSpread Function]

X + *X* + ... + *X*

(**FMINUS** *X*) [Function]

- *X*

(**FDIFFERENCE** *X Y*) [Function]

X - *Y*

(**FTIMES** *X X ... X*) [NoSpread Function]

X * *X* * ... * *X*

(**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 X ... X*) [NoSpread Function]

Returns the minimum of *X* , *X* , . . . , *X* . (FMIN) returns the largest possible floating-point number, the value of MAX.FLOAT.

(**FMAX** *X X ... X*) [NoSpread Function]

Returns the maximum of *X* , *X* , . . . , *X* . (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* . *VALUE*) (*FIELDNAME* . *VALUE*) ...)

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 VALUE FIELDNAME VALUE ...*)

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* *NAME*) is a simple way of defining a *synonym* for the field *NAME*.

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 ... SYN*) *FIELD* must be a field that appears in the current declaration. This defines *SYN ... SYN* 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* 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 ... FORM*)

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 ... FORM* 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 ... COM*) [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 ... COM 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 ITEM ...*) [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 ITEM ...*) [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 ITEM ...*) [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 DATUM*) [Change Word]

Sets *DATUM* to *DATUM* 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 crated.

(**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 contin 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 X ... X*) [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 X ... X*) [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 CLAUSE ... CLAUSE*) [NLambda NoSpread Function]

COND takes an indefinite number of arguments, called clauses. Each *CLAUSE* is a list of the form (*P C ... C*), where *P* is the predicate, and *C ... C* are the consequents. The operation of COND can be paraphrased as:

IF *P* THEN *C ... C* ELSEIF *P* THEN *C ... C* ELSEIF *P ...*

The clauses are considered in sequence as follows: The predicate *P* of the clause *CLAUSE* is evaluated. If the value of *P* is "true" (non-NIL), the consequents *C ... C* are evaluated in order, and the value of the COND is the value of the last expression in the clause. If *P* is "false" (EQ to NIL), then the remainder of *CLAUSE* is ignored, and the next clause, *CLAUSE* , is considered. If no *P* is true for *any* clause, the value of the COND is NIL.

If a clause has no consequents, and has the form (*P*), then if *P* 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 CLAUSE ... CLAUSE DEFAULT*) [NLambda NoSpread Function]

Selects a form or sequence of forms based on the value of *X*. Each clause *CLAUSE* is a list of the form (*S C ... C*) where *S* is the selection key. Think of SELECTQ as:

```
IF X = S THEN C ... C ELSEIF X = S
       THEN ... ELSE DEFAULT
```

If *S* is a symbol, the value of *X* is tested to see if it is EQ to *S* (which is *not* evaluated). If so, the expressions *C ... C* are evaluated in sequence, and the value of the SELECTQ is the value of the last expression.

If *S* is a list, the value of *X* is compared with each element (not evaluated) of *S*, and if *X* is EQ to any one of them, then *C ... C* are evaluated as above.

If *CLAUSE* is not selected in one of the two ways described, *CLAUSE* 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 CLAUSE ... CLAUSE 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 X ... X*) [NLambda NoSpread Function]

Evaluates its arguments in order, and returns the value of its first argument *X*. For example, (PROG1 X (SETQ X Y)) sets X to Y, and returns X's original value.

(**PROG2** *X X ... X*) [NoSpread Function]

Like PROG1. Evaluates its arguments in order, and returns the value of its second argument *X*.

(**PROGN** *X X ... X*) [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 E ... E*) [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 E ... E*) [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 E ... E*) [Macro]
(**PROG*** *VARLST E E ... E*) [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 as J from 1 to N by 2 as K from N to 1 by -1 ...) terminates when I exceeds N , or J exceeds N , 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]

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 ... FORM*). *LAMBDA-WORD* determines whether the arguments to this function will be evaluated or not. *ARG-LIST* determines the number and names of arguments. *FORM ... FORM* 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 ... ARG*) *FORM ... FORM*)

The argument list (*ARG ... ARG*) 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 ... ARG*. Then, *FORM ... FORM* are evaluated in order, and the value of the function is the value of *FORM*.

```
← (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 ... ARG) FORM ... FORM)
```

Nlambda-spread functions are evaluated similarly to lanbda-spread functions, except that the arguments are not evaluated before being bound to the variables *ARG ... ARG*.

```
← (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 ... FORM)
```

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 ... FORM*)

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 . ARGLIST), where ARGLIST is the value SMARTARGLIST should return when *EXPLAINFLG* = T, and ARGLIST the value when *EXPLAINFLG* = NIL. For example, `(GETPROP 'DEFINEQ 'ARGNAMES) = (NIL (X1 X1 ... 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** *X X ... X*) [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* 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* 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 ARG ... ARG*) [NoSpread Function]

Nospread version of APPLY. Applies the function *FN* to the arguments *ARG ARG ... ARG* . 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* . *VAL*) (*NAME* . *VAL*)... (*NAME* . *VAL*))

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 FORM . . . FORM*) [NLambda NoSpread Function]

Nlambda-nospread version of RPT: *N* is evaluated, *FORM* are not. Returns the value of the last evaluation of *FORM* .

(**FRPTQ** *N FORM FORM . . . FORM*) [NLambda NoSpread Function]

Faster version of RPTQ. Does not bind RPTN.

(**MAP** *MAP MAPFN MAPFN*) [Function]

If *MAPFN* is NIL, MAP applies the function *MAPFN* to successive tails of the list *MAP* . That is, first it computes (*MAPFN MAP*), and then (*MAPFN* (CDR *MAP*)), etc., until *MAP* becomes a non-list. If *MAPFN* is provided, (*MAPFN MAP*) is used instead of (CDR *MAP*) for the next call for *MAPFN* , e.g., if *MAPFN* were CDDR, alternate elements of the list would be skipped. MAP returns NIL.

(**MAPC** *MAP MAPFN MAPFN*) [Function]

Identical to MAP, except that (*MAPFN* (CAR *MAP*)) is computed at each iteration instead of (*MAPFN MAP*), i.e., MAPC works on elements, MAP on tails. MAPC returns NIL.

(**MAPLIST** *MAP MAPFN MAPFN*) [Function]

Successively computes the same values that MAP would compute, and returns a list consisting of those values.

(**MAPCAR** *MAP MAPFN MAPFN* [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 MAPFN MAPFN*) [Function]

Computes the same values that MAP and MAPLIST but NCONCs these values to form a list which it returns.

(**MAPCONC** *MAP MAPFN MAPFN*) [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 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*.

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 MAP MAPFN MAPFN*) [Function]

Identical to MAPC except *MAPFN* is a function of two arguments, and (*MAPFN* (CAR *MAP*)(CAR *MAP*)) is computed at each iteration. Terminates when either *MAP* or *MAP* is a non-list.

MAPFN is still a function of one argument, and is applied twice on each iteration; (*MAPFN MAP*) gives the new *MAP*, (*MAPFN MAP*) the new *MAP*. CDR is used if *MAPFN* is not supplied, i.e., is NIL.

(**MAP2CAR** *MAP MAP MAPFN MAPFN*) [Function]

Identical to MAPCAR except *MAPFN* is a function of two arguments, and (*MAPFN* (CAR *MAP*)(CAR *MAP*)) is used to assemble the new list. Terminates when either *MAP* or *MAP* is a non-list.

(**SUBSET** *MAP MAPFN MAPFN*) [Function]

Applies *MAPFN* to elements of *MAP* 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 plays the same role as with MAP, MAPC, et al.

(**EVERY** *EVERY EVERYFN EVERYFN*) [Function]

Returns T if the result of applying *EVERYFN* to each element in *EVERY* is true, otherwise NIL. For example, (EVERY '(X Y Z) 'ATOM) => T.

EVERY operates by evaluating (*EVERYFN* (CAR *EVERY*) *EVERY*). The second argument is passed to *EVERYFN* so that it can look at the next element on *EVERY* if necessary. If *EVERYFN* yields NIL, EVERY immediately returns NIL. Otherwise, EVERY computes (*EVERYFN EVERY*), or (CDR *EVERY*) if *EVERYFN* = NIL, and uses this as the "new" *EVERY*, and the process continues. For example (EVERY X 'ATOM 'CDDR) is true if every other element of *X* is atomic.

(**SOME** *SOME SOMEFN SOMEFN*) [Function]

Returns the tail of *SOME* beginning with the first element that satisfies *SOMEFN*, i.e., for which *SOMEFN* 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* (CAR *SOME*) *SOME*) is computed, and if this not NIL, *SOME* is returned as the value of SOME. Otherwise, (*SOMEFN SOME*) is computed, or (CDR *SOME*) if *SOMEFN* = NIL, and used for the next *SOME*.

(**NOTANY** *SOME SOMEFN SOMEFN*) [Function]

(NOT (SOME *SOME SOMEFN SOMEFN*)).

(**NOTEVERY** *EVERY EVERYFN EVERYFN*) [Function]

(NOT (EVERY *EVERY EVERYFN EVERYFN*)).

(**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 ... X* to these functions are evaluated, though they are not used.

(**NILL** *X ... X*) [NoSpread Function]

Returns NIL.

(**TRUE** *X ... X*) [NoSpread Function]

Returns T.

(**ZERO** *X ... X*) [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.

[This page intentionally left blank]

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 ALINKs 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 CLINKs 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 FOR L)*
—) [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 . PATTERN*) ... (*KEY . PATTERN*)), 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 EXP*)(*VAR EXP*)...). On entering idle mode, each variable *VAR* is bound to the value of the corresponding expression *EXP*. When idle mode is exited, each variable *VAR* 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 FRMAT* —) [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 ... KEY*) [NLambda NoSpread Function]

DATEFORMAT returns a date format suitable as a parameter to DATE and GDATE. *KEY ... KEY* 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 RESOURCE* ...) *FORM FORM* ...) [Macro]

The WITH-RESOURCES macro binds lambda variables of the same name as the resources (for each of the resources *RESOURCE*, *RESOURCE*, etc.) to the result of the GETRESOURCE macro; then executes the forms *FORM*, *FORM*, 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]

MISCELLANEOUS

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).

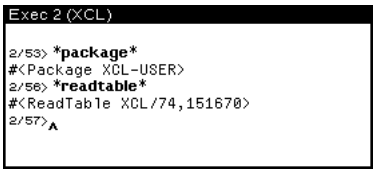
The Exec keeps track of your previous inputs, in the history list. Each entry you type creates a history event, which sotres the input and its values.

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

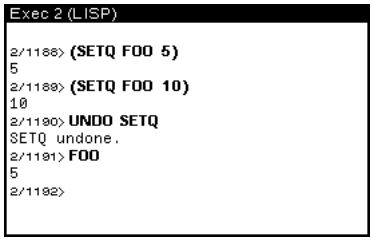
Sometimes you need more than one Exec open at a time. It's easy to open as many as you need by using the right button background menu and selecting the kind of Exec you need. The Execs are differentiated from one another by their "names" in their title bars and by their prompts. For example, the second 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:

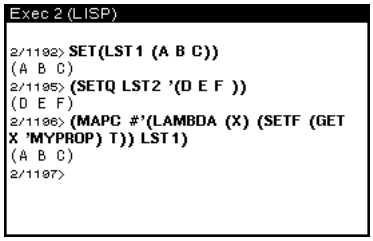


A Brief Example of Exec Interactions

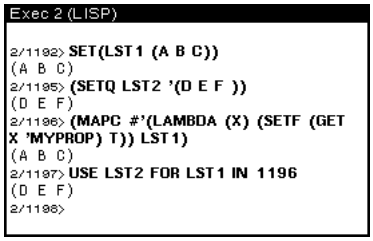
The following dialogue contains examples and gives the flavor of the use of an Exec. The commands are described in greater detail in the following sections. For now, be sure to type these examples to an Exec whose *PACKAGE* is set to the XCL-USER package. The Exec that Medley starts up with is set to 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.

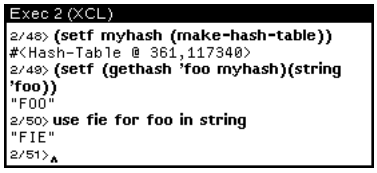


The Exec accepts input both in APPLY format (the SET) and EVAL format (the SETQ). In event 1196, you added a property MYPROP to the symbols A, B, and C.

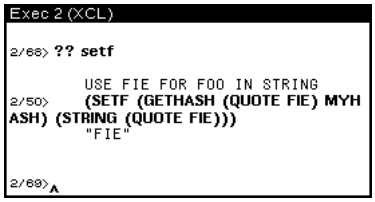


You told the Exec to go back to event 1196, substitute LST2 for LST1, and then re-execute the expression.

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.

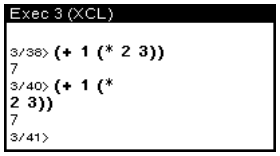
Input Formats

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 1 not 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* AND *EventAddress*, the notation *EventAddress* corresponds to all words up to the AND in the event specification, and *EventAddress* 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 AND *EventSpec* AND . . . AND *EventSpec*

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* **FOR** *OLD* **AND** ... **AND** *NEW* **FOR** *OLD* [**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]

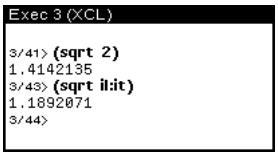
Prints the contents of *FILES* in the Exec window, hiding comments.

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]

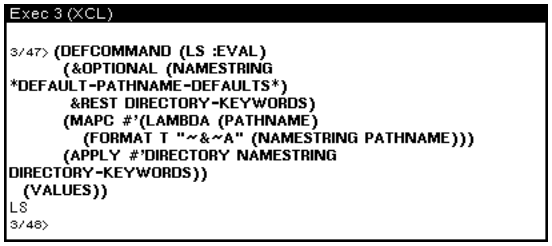
XCL:DEFCOMMAND is like XCL:DEFMACRO, but defines new Exec commands. The *ARGUMENT-LIST* can have keywords, and use all of the features of macro argument lists. When *NAME* is subsequently typed to the Exec, the rest of the line is processed like the arguments to a macro, and the *BODY* is executed. XCL:DEFCOMMAND 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),



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 VALUE PROPERTY VALUE ...), 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.

[This page intentionally left blank]

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.

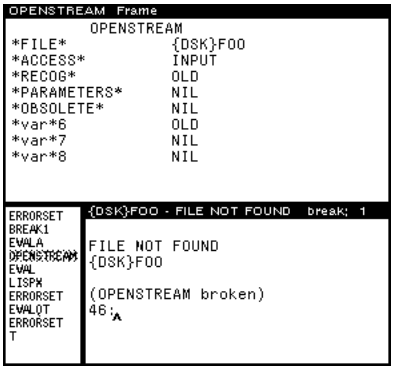
Break Windows

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 COM1 ... COM1*)(*NAME COM2 ... COM2*)...). Whenever an atomic command is given to BREAK1, it first searches the list BREAKMACROS for the command. If the command is equal to *NAME* , 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 MESS NOBREAK*) [Function]

Prints *MESS* (using PRIN1), followed by a space if *MESS* is an atom, otherwise a carriage return. Then *MESS* is printed (using PRIN1 if *MESS* 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* and *MESS* 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* . *MESS*))), 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 MESS 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 MESS BRKTYPE*) [Function]

Prints *MESS* and *MESS* similar to ERROR, and then calls BREAK1 passing *BRKTYPE* as the BRKTYPE argument. If both *MESS* and *MESS* 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 MESS MESS*) [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 ... FORM*) [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* , 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 E ... E*) [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 FORM ... FORM*) [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 FORM* **...** *FORM*). For example, (RESETFORM (RADIX 8) (FOO)). RESETFORM compiles open. If no error occurs, it returns the value returned by *FORM* .

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 thes 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* IN *FN*), BREAK0 breaks every call to *FN* from within *FN* . 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* is compiled or blockcompiled, whereas BREAKIN only works on interpreted functions. If *FN* is not found in *FN* , BREAK0 returns the value (*FN* NOT FOUND IN *FN*).

BREAK0 breaks one function *inside* another by first calling a function which changes the name of *FN* wherever it appears inside of *FN* to that of a new function, *FN1*-IN-*FN2*,

which is initially given the same function definition as *FN*. 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* and puts (*FN* . *FN*) 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* 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* IN *FN*), (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* IN *FN*)) is allowed: UNBREAK0 will operate on (*FN* -IN-*FN*) 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* IN *FN*), *FN* is changed to *FN1*-IN-*FN2* throughout *FN* , as with break, and then *FN1*-IN-*FN2* is used in place of *FN*. If *FN* and/or *FN* 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:

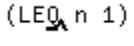
The structure caret appears when the edit point is between symbols (or strings or comments), so that anything you type will go into a new one. It looks like this:

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

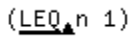
The left mouse button selects parts of Lisp structures. The middle mouse button selects whole Lisp structures.

For example; select the `Q` in `LEQ` below by pressing the left mouse button when the pointer is over the `Q`.



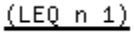
Any characters you type in now will be appended to the symbol LEQ.

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:



Any characters you type in now will form a new symbol between the LEQ and the n.

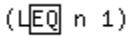
Larger structures can be selected in two ways. Use the middle mouse button to position the mouse cursor on the parenthesis of the list you want to edit. Press the mouse button multiple times, without moving the mouse, extends the selection. In the previous example, if the middle button was pressed twice, the list (LEQ . . .) would be selected:



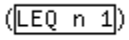
Press the button a third time and you will select the list containing the (LEQ n 1) to be selected.

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:



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:



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: ' 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.

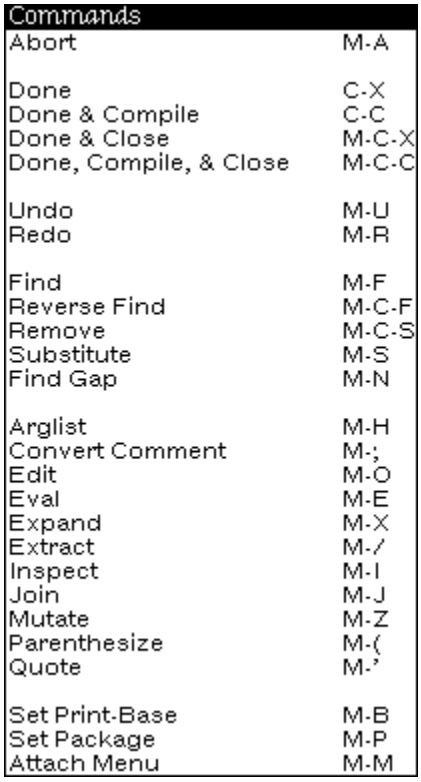
Meta-M Attach a menu of common commands to the top of the SEdit window. Each SEdit window can have its own menu.

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

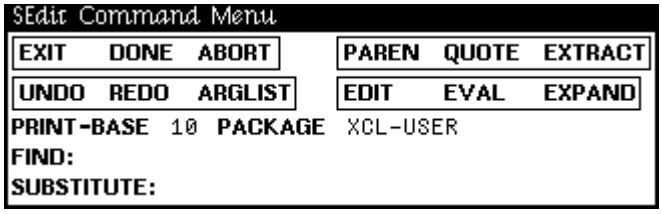
When the mouse cursor is in the SEdit title bar and you press middle mouse button, a Help Menu of commands pops up. The menu looks like this:



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 familliar 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 ... PATTERN*) [Editor Command]

Equivalent to F *PATTERN* followed by F *PATTERN* ... followed by F *PATTERN* , so that if F *PATTERN* fails, the edit chain is left at the place *PATTERN* matched.

(F= *EXPRESSION X*) [Editor Command]

Equivalent to (F (== . *EXPRESSION*) *X*), i.e., searches for a structure EQ to *EXPRESSION* (see above).

(ORF *PATTERN ... PATTERN*) [Editor Command]

Equivalent to (F (*ANY* *PATTERN ... PATTERN*) N), i.e., searches for an expression that is matched by either *PATTERN* , *PATTERN* , ... or *PATTERN* (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 MARKs 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* ... *E*) (*N* >= 1) [Editor Command]

Replaces the *N*th element in the current expression with *E* ... *E* .

(-*N E* ... *E*) (*N* >= 1) [Editor Command]

Inserts *E* ... *E* before the *N*th element in the current expression.

(N *E* ... *E*) [Editor Command]

Attaches *E* ... *E* 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 ... E*), and (*-N E ... E*) 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 ... E*) [Editor Command]

Inserts *E ... E* before the current expression. Equivalent to UP followed by (-1 *E ... E*).

For example, to insert FOO before the last element in the current expression, perform -1 and then (B FOO).

(A *E ... E*) [Editor Command]

Inserts *E ... E* after the current expression. Equivalent to UP followed by (-2 *E ... E*) or (N *E ... E*), whichever is appropriate.

(: *E ... E*) [Editor Command]

Replaces the current expression by *E ... E*. Equivalent to UP followed by (1 *E ... E*).

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* ... *E* BEFORE . @) [Editor Command]

(@ is (CDR (MEMBER 'BEFORE *COMMAND*))) Similar to (LC .@) followed by (B *E* ... *E*).

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* ... *E* AFTER . @) [Editor Command]

Similar to INSERT BEFORE except uses A instead of B.

(INSERT *E* ... *E* FOR . @) [Editor Command]

Similar to INSERT BEFORE except uses : for B.

(REPLACE @ BY *E* ... *E*) [Editor Command]
(REPLACE @ WITH *E* ... *E*) [Editor Command]

Here @ is the *segment* of the command between REPLACE and WITH. Same as (INSERT *E* ... *E* FOR . @).

Example: (REPLACE COND -1 WITH (T (RETURN L)))

(CHANGE @ TO *E* ... *E*) [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* thru *E* 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 @ FROM . @) [Editor Command]

Performs (LC . @) and then (XTR . @) (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: @ 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* ... *E*) [Editor Command]

MBD substitutes the current expression for all instances of the atom & in *E* ... *E* , 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 ... E*, the MBD is interpreted as (MBD (*E ... E* &)).

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 @ TO *COM* . @) [Editor Command]

(@ is the segment between MOVE and TO.) *COM* is BEFORE, AFTER, or the name of a list command, e.g., :, N, etc. Performs (LC . @) (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 . @) followed by (*COM EXPR*) (setting an internal flag so *EXPR* is not copied), then goes back to @ and deletes *EXPR*. The edit chain is not changed. UNFIND is set to the edit chain after (*COM EXPR*) was performed.

If @ 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 @ , 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 @ 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., @ . 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 @ 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.

(@ THRU @) [Editor Command]

Does a (LC . @), followed by an UP, and then a (BI 1 @), 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)).

(@ TO @) [Editor Command]

Same as THRU except the last element not included, i.e., after the BI, an (RI 1 -2) is performed.

If both @ and @ are numbers, and @ is greater than @ , then @ counts from the beginning of the current expression, the same as @ . 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 @ -@ +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.

(@ TO) [Editor Command]
(@ THRU) [Editor Command]

Both are the same as (@ THRU -1), i.e., from @ 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 @ @) [Editor Command]

Like SW except switches the expressions specified by @ and @ , not the corresponding elements of the current expression, i.e. @ and @ 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))).

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 CONDs together, e.g. (COND *CLAUSE CLAUSE*) followed by (COND *CLAUSE CLAUSE*) becomes (COND *CLAUSE CLAUSE CLAUSE CLAUSE*). 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 CLAUSE CLAUSE CLAUSE*) into (COND *CLAUSE CLAUSE*) (COND *CLAUSE CLAUSE*). 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* and *N* are supplied, (*N* THRU *N*) is used rather than the current expression; if just *N* is supplied, (*N* 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.

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 X ... X*) [Editor Command]

Executes the *editor command* (*C Y ... Y*) where *Y* = (EVAL *X*). 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.

(## *COM* *COM* ... *COM*) [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* ... *COM* starting from the present edit chain. Generates an error if any of *COM* thru COM 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* ... *X*) [Editor Command]

Each *X* 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 ... COM*) [Editor Command]

Executes *COM ... COM* .

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 COMS*) [Editor Command]

If (EVAL *X*) is true, execute *COMS* ; if (EVAL *X*) causes an error or is equal to NIL, execute *COMS* .

Thus IF is equivalent to

```
(COMS (CONS 'COMSQ
        (COND
           ((CAR (NLSETQ (EVAL X)))
              COMS )
           (T  COMS ))))
```

For example, the command (IF (READP T) NIL (P)) will print the current expression provided the input buffer is empty.

(IF *X COMS*) [Editor Command]

If (EVAL *X*) is true, execute *COMS* ; otherwise generate an error.

(LP *COMS ... COMS*) [Editor Command]

Repeatedly executes *COMS ... COMS* 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 ... COMS* .

(LPQ *COMS ... COMS*) [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 ... COMS*) [Editor Command]

ORR begins by executing *COMS* , 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* , 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 ... COMS*) [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 ... COMS* .

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* ... *ARG*) *COMS* ... *COMS*) [Editor Command]

C an atom. M defines *C* as a list command. Executing (*C E* ... *E*) is then performed by substituting *E* for *ARG*, ... *E* for *ARG* throughout *COMS* ... *COMS*, and then executing *COMS* ... *COMS*.

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* ... *COMS*) [Editor Command]

If *C*, *ARG* are both atoms, this defines *C* as a list command. Executing (*C E* ... *E*) is performed by substituting (*E* ... *E*), i.e., CDR of the command, for *ARG* throughout *COMS* ... *COMS*, and then executing *COMS* ... *COMS*.

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 ... COMS*) [Editor Command]

Binds three dummy variables #1, #2, #3, (initialized to NIL), and then executes the edit commands *COMS ... COMS* . 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 ... COMS*) [Editor Command]

Executes *COMS ... COMS* 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 ... FILE*) [NLambda NoSpread Function]

Loads the files *FILE ... FILE* (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 FILE ... FILE*) [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* 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 FILE ... FILE*) [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 FILE ... FILE*) [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* 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 explicity 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?

andCENPwl rn ann esgs n hncl DTFLS opri [redacted]t pcf nwihflsteeiesbln.[redacted] a loivk PAEIE iety cify 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 NAME TYPE SOURCE SOURCE*) [Function]

Compares the definition of *NAME* with that of *NAME* , by calling COMPARELISTS (Chapter 3) on (GETDEF *NAME TYPE SOURCE*) and (GETDEF *NAME TYPE SOURCE*), 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 VAL ... PROP VAL*) [NoSpread Function]

Nospread function for defining new file manager types, or changing properties of existing file manager types. *PROP* is one of the property names given above; *VAL* 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 ... SYMBOL*) [File Manager Command]

Each symbol *SYMBOL* is the name of a user edit macro. Writes expressions to add the edit macro definitions of *SYMBOL* to USERMACROS, and adds the names of the commands to the appropriate spelling lists.

If *SYMBOL* is not a user macro, a warning message "no EDIT MACRO for *SYMBOL* " is printed.

Functions and Macros

(**FNS** *FN ... FN*) [File Manager Command]

Writes a DEFINEQ expression with the function definitions of *FN ... FN* .

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 ... FN*) [File Manager Command]

For each function *FN* , 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 ... FN*) [File Manager Command]

For each function *FN* , 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 ... SYMBOL*) [File Manager Command]

Each *SYMBOL* 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* , embedded in a DECLARE: EVAL@COMPILE so the macros will be defined when the file is compiled. See Chapter 10.

Variables

(**VARS** *VAR ... VAR*) [File Manager Command]

For each *VAR* , writes an expression to set its top level value when the file is loaded. If *VAR* is atomic, VARS writes out an expression to set *VAR* to the top-level value it had at the time the file was written. If *VAR* 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 ... VAR*) [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* 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 . LST*)...(*VAR . LST*)) [File Manager Command]

For each (*VAR . LST*), writes an ADDTOVAR (see the Functions Used Within Source Files section) to add each element of *LST* to the list that is the value of *VAR* at the time the file is loaded. The new value of *VAR* will be the union of its old value and *LST* . If the value of *VAR* 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* is not specified, *VAR* 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* . *LST*) ... (*VAR* . *LST*)) [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* ... *VAR*) [File Manager Command]

Like VARS, except that the value of each *VAR* 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* ... *VAR*) [File Manager Command]

Like UGLYVARS, except structures may also contain circular pointers. Uses HPRINT (see Chapter 25). The values of *VAR* ... *VAR* 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 KEY KEY* ...)...(*VAR KEY KEY* ...)) [File Manager Command]

VAR is a variable whose value is an association list, such as EDITMACROS, BAKTRACELST, etc. For each *VAR*, 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* ... *VAR*) [File Manager Command]
(**LOCALVARS** *VAR* ... *VAR*) [File Manager Command]
(**GLOBALVARS** *VAR* ... *VAR*) [File Manager Command]

Outputs the corresponding compiler declaration embedded in a DECLARE: DOEVAL@COMPILE DONTCOPY. See Chapter 18.

(**CONSTANTS** *VAR* ... *VAR*) [File Manager Command]

Like VARS, for each *VAR* 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* 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 ... SYMBOL*) [File Manager Command]

Writes a PUTPROPS expression to restore the value of the *PROPNAME* property of each symbol *SYMBOL* 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*) 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* does not have the property *PROPNAME* (as opposed to having the property with value NIL), a warning message "NO *PROPNAME* PROPERTY FOR *SYMBOL* " 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 ... SYMBOL*) [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 PROPNAME*)...(*SYMBOL PROPNAME*)) [File Manager Command]

Similar to PROP command. Writes a PUTPROPS expression to restore the value of *PROPNAME* for each *SYMBOL* when the file is loaded.

As with the PROP command, if *SYMBOL* does not have the property *PROPNAME* (as opposed to having the property with NIL value), a warning message "NO *PROPNAME* PROPERTY FOR *SYMBOL* " is printed.

Miscellaneous File Manager Commands

(**RECORDS** *REC ... REC*) [File Manager Command]

Each *REC* is the name of a record (see Chapter 8). Writes expressions which will redeclare the records when the file is loaded.

(**INITRECORDS** *REC ... REC*) [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 ... SYMBOL*) [File Manager Command]

Each *SYMBOL* 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 ... OPR*) [File Manager Command]

Each *OPR* 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 ... RESOURCE*) [File Manager Command]

Each *RESOURCES* is the name of a resource (see Chapter 12). Writes expressions which will redeclare the resource when the file is loaded.

(**INITRESOURCES** *RESOURCE ... RESOURCE*) [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 ... NAME*) [File Manager Command]

Each *NAME* 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 ... SYMBOL*) [File Manager Command]

Each *SYMBOL* 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 ... FILE*) [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 ... FILE* as the arguments. FILESLOAD interprets *FILE ... FILE* as files to load, possibly interspersed with lists used to specify certain loading options.

(**FILEPKGCOMS** *SYMBOL ... SYMBOL*) [File Manager Command]

Each symbol *SYMBOL* 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* is not a file manager command or type, a warning message "no FILE PACKAGE COMMAND for *SYMBOL* " 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 ... EXP*) [File Manager Command]

Writes each of the expressions *EXP ... EXP* on the output file, where they will be evaluated when the file is loaded.

(**E** *FORM ... FORM*) [File Manager Command]

Each of the forms *FORM ... FORM* is evaluated at *output* time, when MAKEFILE interpretes this file manager command.

(**COMS** *COM ... COM*) [File Manager Command]

Each of the commands *COM ... COM* is interpreted as a file manager command.

(**ORIGINAL** *COM ... COM*) [File Manager Command]

Each of the commands *COM* 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 ... BLOCK*) [File Manager Command]

For each *BLOCK* , 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 ... COM*) [File Manager Command]

This command is used for "exporting" definitions. Like COM, each of the commands *COM ... COM* 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 VAL ... PROP VAL*) [NoSpread Function]

Nospread function for defining new file manager commands, or changing properties of existing file manager commands. *PROP* is one of of the property names described above; *VAL* 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 ar 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 X ... X*) [NLambda NoSpread Function]

Each *X* 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 X ... X*) *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 X ... X*) [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 VAL ... PROP VAL*) [NLambda NoSpread Function]

Nlambda nospread version of PUTPROP (none of the arguments are evaluated). For *i* = 1...*N*, puts property *PROP* , 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* ... *ARG*) (uses: *VAR* ... *VAR*) (calls: *FN* ... *FN*))

The message is printed at the beginning of the second pass of the compilation of *FN*. (*ARG* ... *ARG*) 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 BLKFN* ...). 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 NILs, 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 VAR ... VAR*) [NLambda NoSpread Function]

Defines *VAR* , ... *VAR* (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*) instead.

If *VAR* 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 ... BLOCK )
```

where each *BLOCK* 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 ... BLKFN
  (VAR . VALUE ) ... (VAR . VALUE ))
```

BLKNAME is the name of a block. *BLKFN ... BLKFN* are the functions in the block and correspond to *BLKFNS* in the call to BLOCKCOMPILE. The (*VAR . VALUE*) expressions indicate the settings for variables affecting the compilation of that block. If *VALUE* is atomic, then *VAR* is set to *VALUE* , otherwise *VAR* is set to the UNION of *VALUE* and the current value of the variable *VAR* . 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 ... BLKFN* (*VAR . VALUE*) ... (*VAR . VALUE*))

which means that *BLKFN ... BLKFN* should be compiled after first setting *VAR ... VAR* 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* ...), 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* ... *VAL*).

(*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, $\leftarrow$*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 $\leftarrow$ followed by a form.

Note that an atom of the form X$\leftarrow$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$\leftarrow$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$\leftarrow$(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 ... FN*) [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*), 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 ... FN*) [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 ... VAR*).

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 ... VAR*)) 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 ... FN*) [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* **...** *VAR*).

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* **...** *VAR*)) 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 ... FN*) [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 VALUE ... PROP VALUE*) [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:

(**CREAT.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 ... FORM*) [Macro]

Evaluates *FORM ... FORM* while owning *LOCK*, and returns the value of *FORM* . 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 ... FORM* .

(**WITH.FAST.MONITOR** *LOCK FORM ... FORM*) [Macro]

Like WITH.MONITOR, but implemented without the RESETLST. User interrupts (e.g., Control-E) are inhibited during the evaluation of *FORM ... FORM* .

Programming restriction: the evaluation of *FORM ... FORM* 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.

The following resources, which are global in Interlisp-10, are allocated per proces 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.

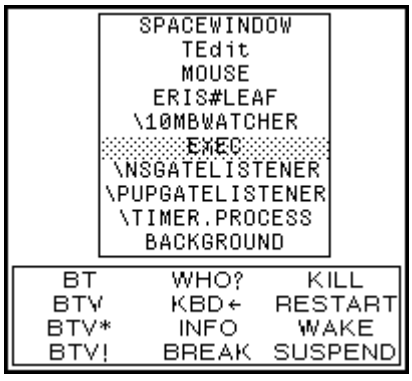
(**ALLOW.BUTTON.EVENTS**) [Function]

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 VALUE ... PROP VALUE*) [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 releass 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* ... *FORM*) [Macro]

Evaluates *FORM* ... *FORM* while owning *LOCK*, and returns the value of *FORM* . 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* ... *FORM* .

(WITH.FAST.MONITOR *LOCK FORM* ... *FORM*) [Macro]

Like WITH.MONITOR, but implemented without the RESETLST. User interrupts (e.g., Control-E) are inhibited during the evaluation of *FORM* ... *FORM* .

Programming restriction: the evaluation of *FORM* ... *FORM* 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, an 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—ar 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.

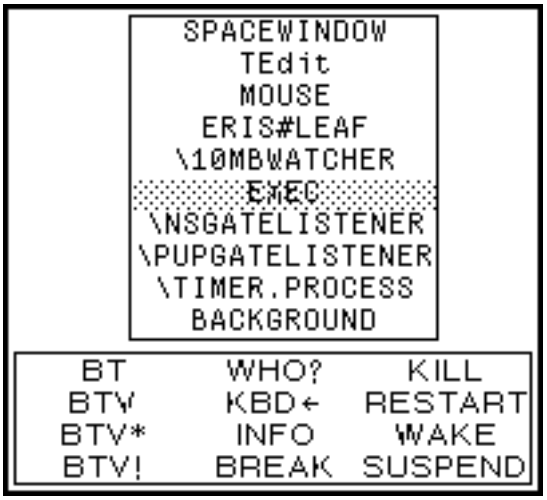
(ALLOW.BUTTON.EVENTS) [Function]

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 director 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 DIRECTOR ils
lds.

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 CONTENTS ... FIELD CONTENTS*) [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 CONTENTS ... FIELD CONTENTS*) [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 isundoable. —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 STRPT)*

[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 VAL ... PROP VAL*) [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 ... COM*) [NLambda NoSpread Function]

Convenient form of DIRECTORY for use in type-in at the executive. Performs (DIRECTORY '*FILEGROUP* '(P *COM ... COM*)).

(**NDIR** *FILEGROUP COM ... COM*) [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 perfrm 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 CONTENTS ... FIELD CONTENTS*) [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 CONTENTS ... FIELD CONTENTS*) [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 connectd 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, STREAMs 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 VAL ... PROP VAL*) [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     READDA
           TE WRITEDATE
```

(FILDIR *FILEGROUP*) [Function]

Obsolete synonym of (DIRECTORY *FILEGROUP*).

(DIR *FILEGROUP COM ... COM*) [NLambda NoSpread Function]

Convenient form of DIRECTORY for use in type-in at the executive. Performs (DIRECTORY '*FILEGROUP* '(P *COM ... COM*)).

(NDIR *FILEGROUP COM ... COM*) [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 beak 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 ... PRINTCOM*)

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 ... OPTION 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 ... OPTION* , 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 VAL) ... (PROP VAL)).
As an ALIST	Inspects the list as an association-list, if the list is in ASSOC list form: (PROP VAL ... PROP VAL).
As a record	Brings up a submenu with all of the RECORDs 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 ... FN*) [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)      (COMMEN
T1 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 REGION ... REGION*) [NoSpread Function]

Returns a region which is the intersection of a number of regions. Returns NIL if the intersection is empty.

(**UNIONREGIONS** *REGION REGION ... REGION*) [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 REGION*) [Function]

Returns T if *REGION* intersects *REGION* . 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.

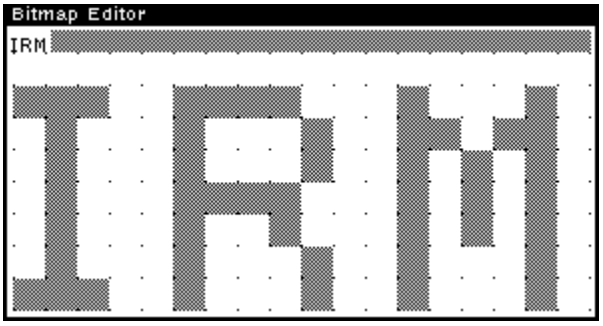
(**READBITMAP** *FILE*) [Function]

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).

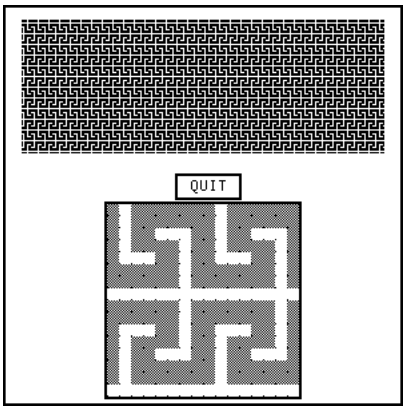
The common textures white and black are available as system constants WHITESHADE and BLACKSHADE. The global variable GRAYSHADE is used by many system facilities as a background gray shade and can be set by the user.

(**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 acepted 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 strems. 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 VAL PROP VAL ...) where PROP is acceptable as a font-property to FONTCOPY, then the new font is obtained by (FONTCOPY (DSPFONT NIL STREAM) PROP VAL PROP VAL ...). 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:

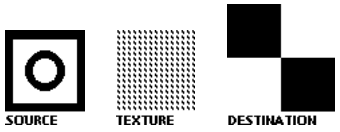
REPLACE All source bits (on or off) replace destination bits.

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.

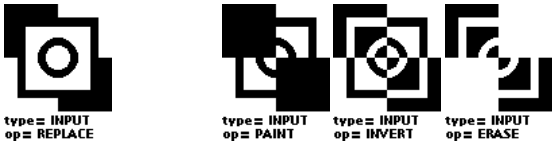
INVERT Any source bits that are on invert the corresponding destination bits. Does a logical XOR between the source bits and the destination bits.

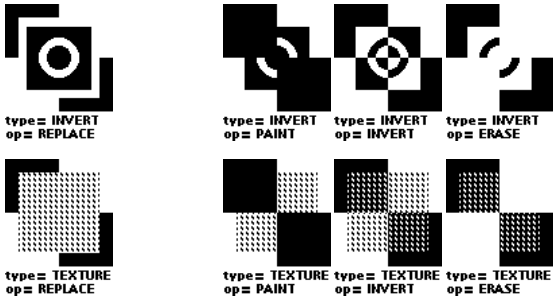
ERASE Any source bits that are on erase the corresponding destination bits. Does a logical AND operation between the inverse of the source bits and the destination bits.

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 Y X Y WIDTH OPERATION STREAM COLOR DASHING*) [Function]

Draws a straight line from the point (*X* ,*Y*) to the point (*X* ,*Y*) on the image stream *STREAM*. The position of *STREAM* is set to (*X* ,*Y*). If *X* equals *X* and *Y* equals *Y* , a point is drawn at (*X* ,*Y*).

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 POSITION WIDTH OPERATION STREAM COLOR DASHING*) [Function]

Draws a line from the point *POSITION* to the point *POSITION* onto the destination bitmap of *STREAM*. The position of *STREAM* is set to *POSITION* .

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]

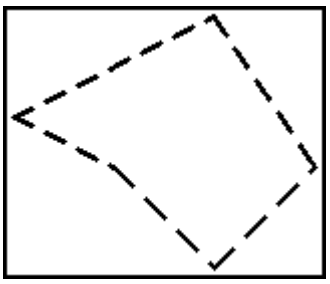
Draws a polygon on the image stream *STREAM*. *POINTS* is a list of positions to which the figure will be fitted (the vertices of the polygon). If *CLOSED* is non-NIL, then the starting position is specified only once in *POINTS*. If *CLOSED* is NIL, then the starting vertex must be specified twice in *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:

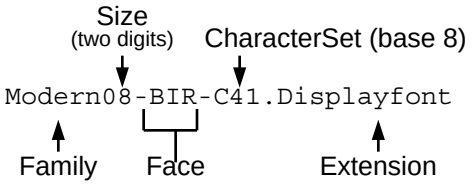
```
STANDARD = (MEDIUM REGULAR REGULAR) = MRR
ITALIC = (MEDIUM ITALIC REGULAR) = MIR
BOLD = (BOLD REGULAR REGULAR) = BRR
BOLDITALIC = (BOLD ITALIC REGULAR) = BIR
```

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 VAL PROP VAL* ...) [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 VAL PROP VAL ...* 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 dta 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. Anyne 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 Y X Y WIDTH OPERATION COLOR DASHING*) [Image Stream Method]

Draws a line of width *WIDTH* from (*X* , *Y*) to (*X* , *Y*). 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.

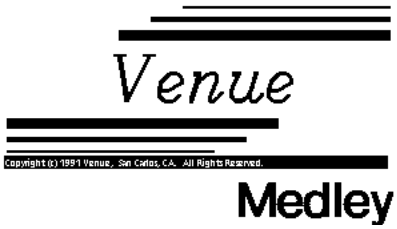
Warning: The window system assumes that all programs follow certain conventions concerning control of the screen. All user programs should use perform display operations using windows and menus. In particular, user programs should not perform operate directly on the screen bitmap; otherwise the window 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 ... EXP*) [NoSpread Function]

Clears the prompt window, and prints *EXP* through *EXP* 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.

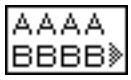
An item with subitems is marked in the menu by a grey, right pointing triangle following the label.

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:

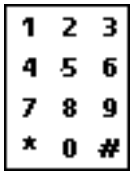
The strings following the YES and NO are help strings and will be printed if the cursor remains over one of the items for a period of time. This menu differs from the one above in that it distinquishes the NO case from the case where the user clicked outside of the menu. If the user clicks outside of 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)))
```

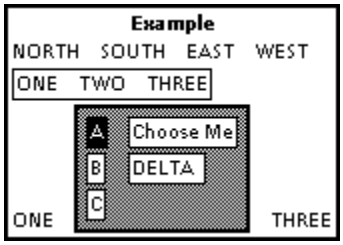
This menu has two command buttons, LOOKUP and EXIT, and three edit fields, with IDs NAME, PHONE, and ADDRESS. The Edit items are initialized to the empty string, as in this example they need no other initial value. The user could select the Name: prompt, type a person's name, and then press the LOOKUP 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
```

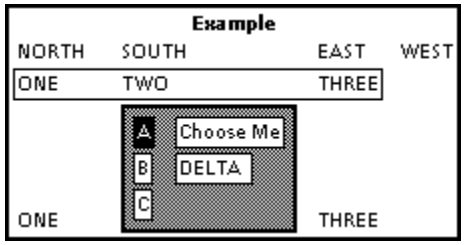
Here is a more complicated example:

```
((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))
```

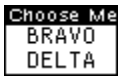
The three items, labelled A, B, and C are all declared as NWAY items, and are also specified to belong to the same NWAY Collection, Col1. This is how a number of NWAY items are collected together. The property NWAYPROPS (DESELECT T) on the first NWAY item specifies that the Col1 Collection is to have 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 don e 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.

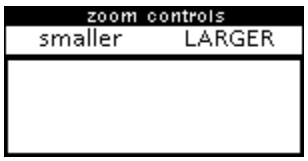
Examples:

```
(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 VALUE PROP VALUE ...). 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  VALUE ) (PROPERTY  VALUE )
...)
```

TYPES is a list of the printer types that this entry addresses. The (PROPERTY VALUE) 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 , FORM ,... FORMN* [NLambda NoSpread Function]

Clears any typeahead (ringing the terminal's bell if there was, indeed, typeahead), evaluates *FORM* , *FORM ,... FORM* , then restores the typeahead. Returns the value of *FORM* . 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.

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).

(**KYACTION** *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*.

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.

CLISP	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 (PRlNT 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.

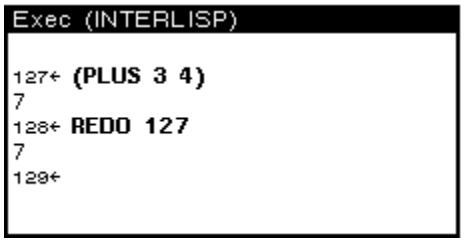


Figure 2-1. Using a Programmer's Assistant Command to `REDO` a Function

- 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.

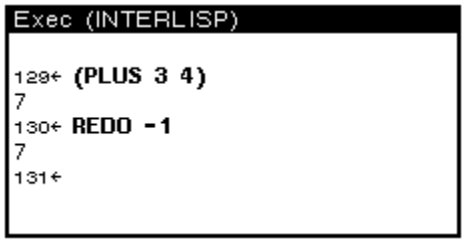


Figure 2-2. Using a Negative Number after the Programmer's Assistant Command

- 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.

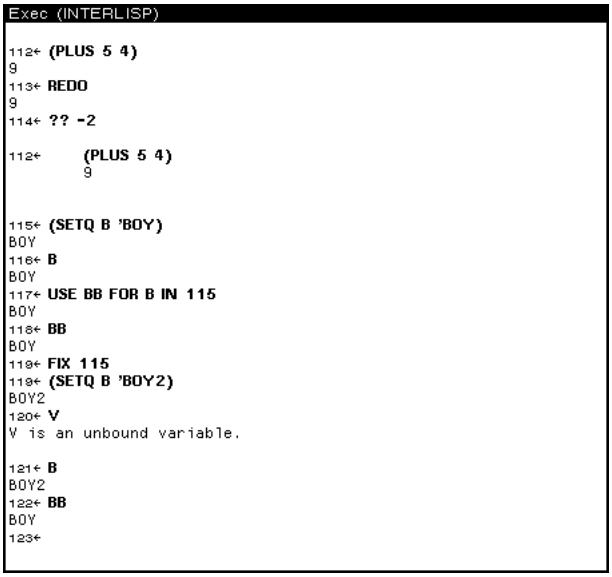


Figure 2-3. Some Applications of the Programmer's Assistant

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

21. The Grapher

22. Resource Management

23. Simple Interactions with the Cursor, a Bitmap, and a Window

24. Glossary of Global System Variables

25. Other Useful References

Index

[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.

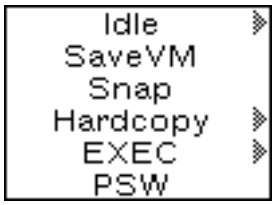


Figure 3-1. Background Menu

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.

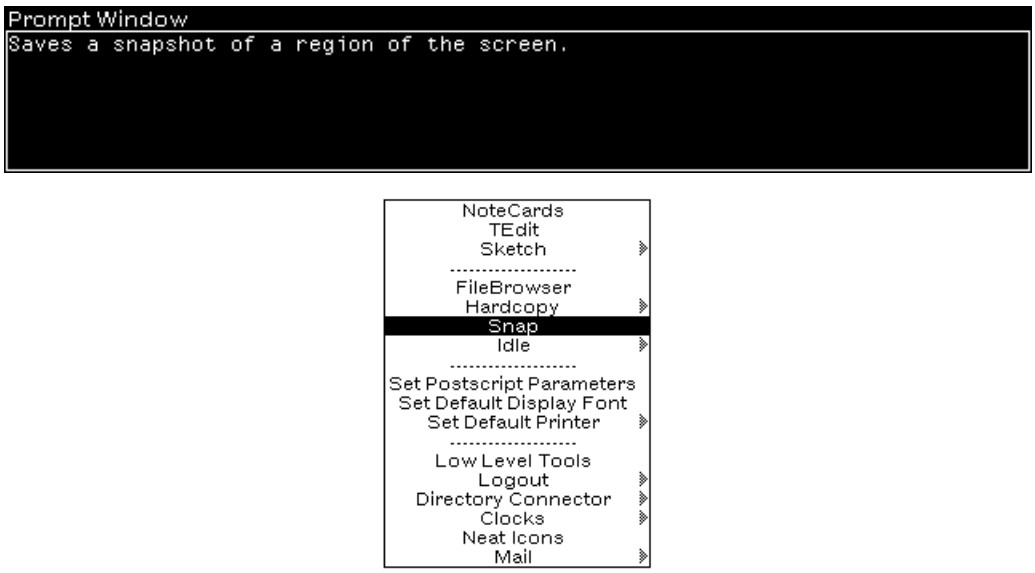


Figure 3-4. Explanation Associated with Selected Menu Item

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 submen 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.

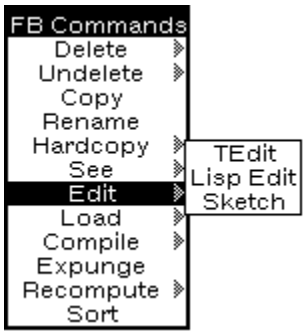


Figure 3-5. Edit Submenu Displayed with Right Arrow

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 utton 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.

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.

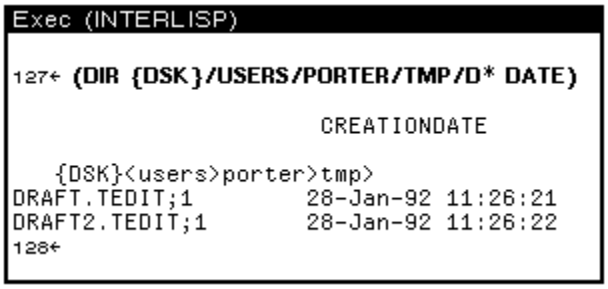


Figure 4-2. Example Using DATE

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*)

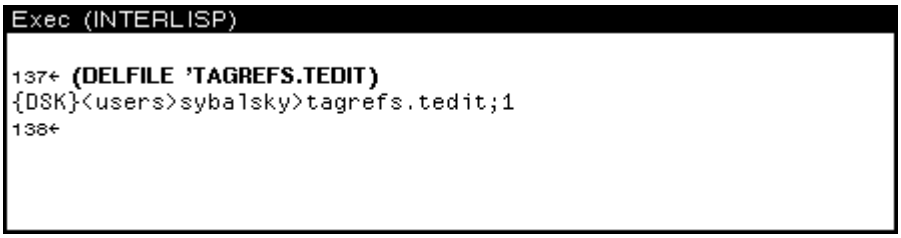


Figure 4-4. Example Use of DELFILE

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.

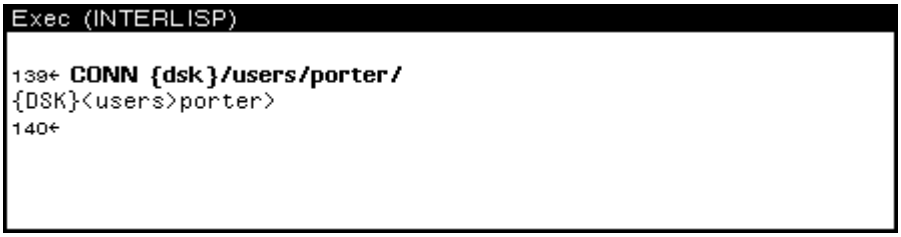


Figure 4-5. CONNecting to Subdirectory Primer Subsubdirectory IM

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.

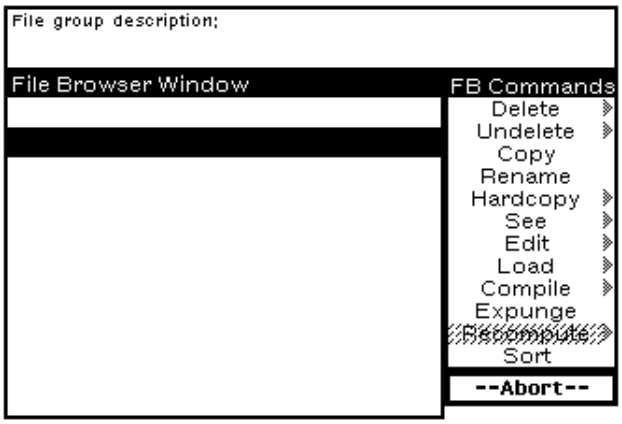


Figure 5-1. Prompt for Files to Include in FileBrowser

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.

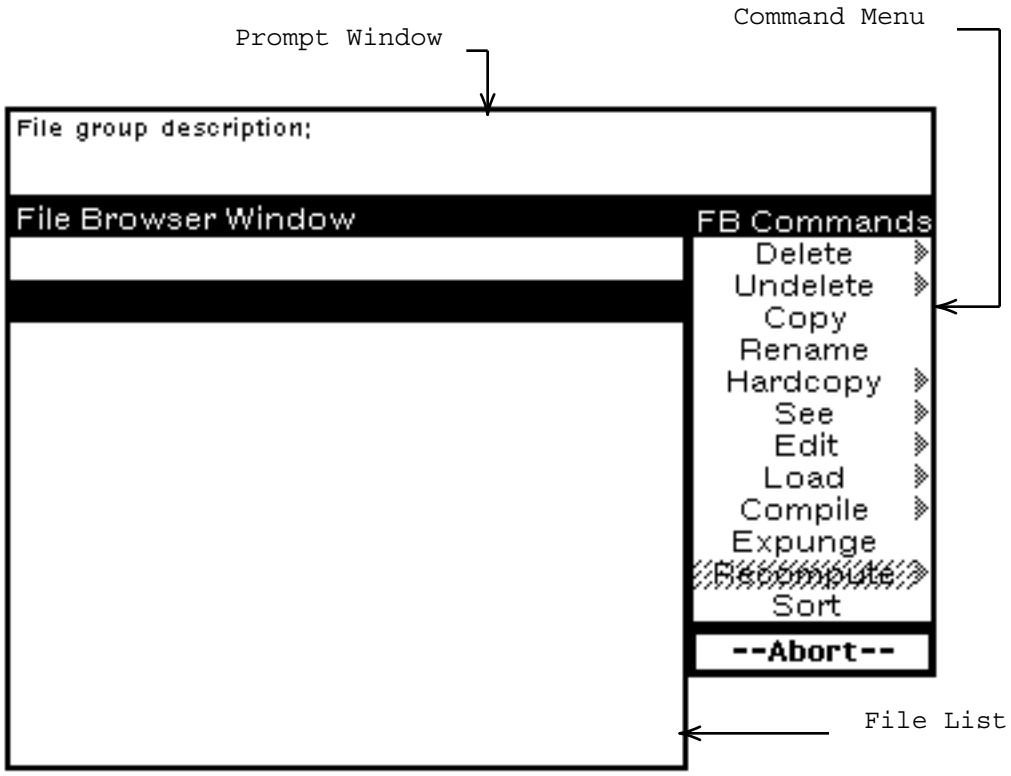


Figure 5-2. Parts of a FileBrowser Window

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.

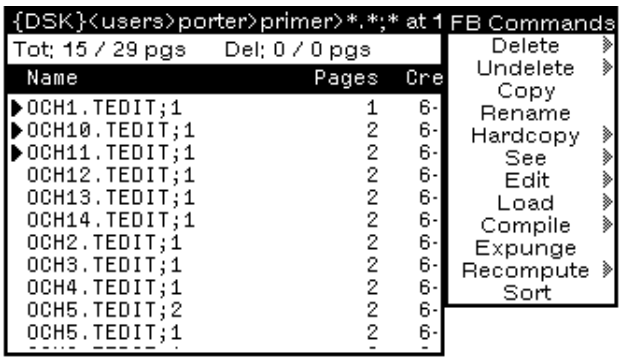


Figure 5-3. Files Chosen

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.

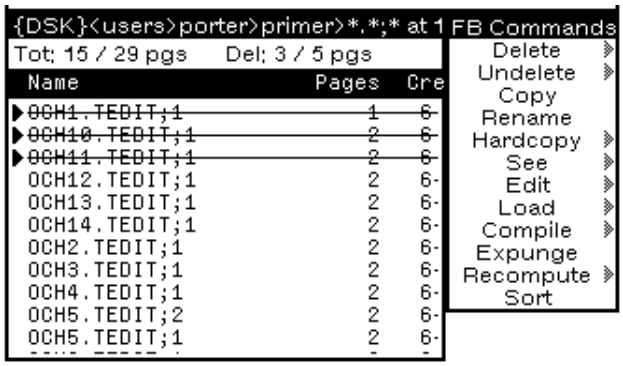


Figure 5-4. Files Marked for Deletion

`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.

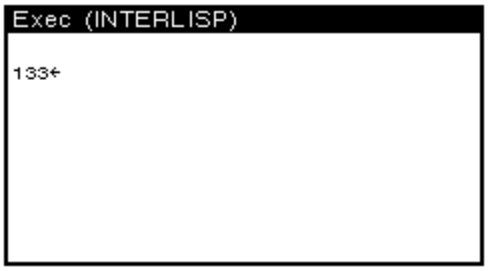


Figure 6-1. Medley Executive Window

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.)

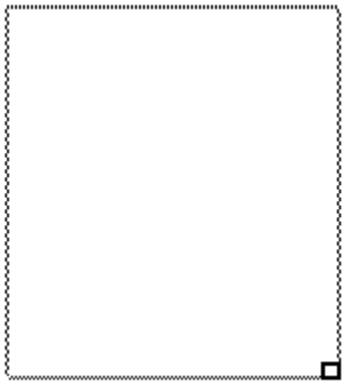


Figure 6-3. Mouse Cursor Asking You to Sweep Out Window

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.)

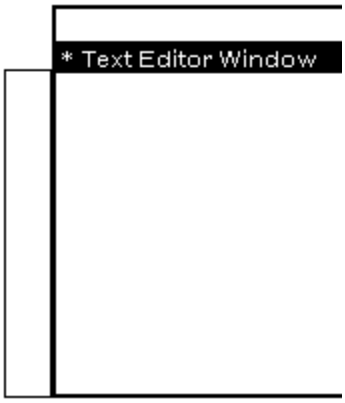


Figure 6-7. Scroll Bar of Scrollable Window

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.

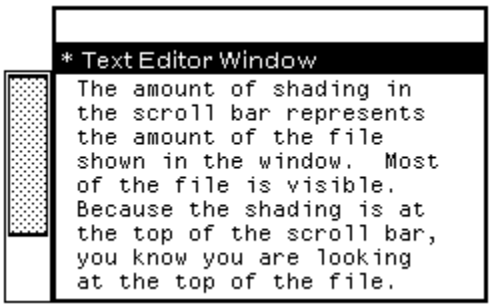


Figure 6-8 Top of File When Shading at Top of Scroll Bar

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.

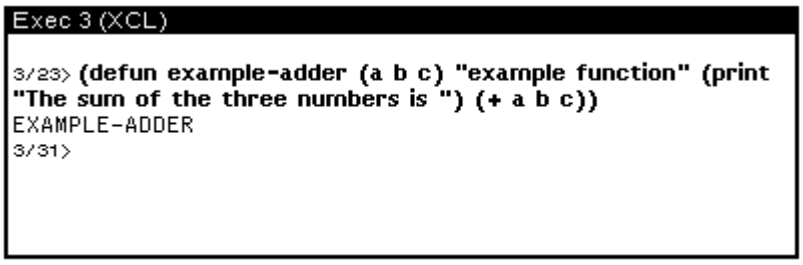


Figure 7-1. Defining the Function EXAMPLE-ADDER

Now that the function is defined, it can be called from the Executive Window:

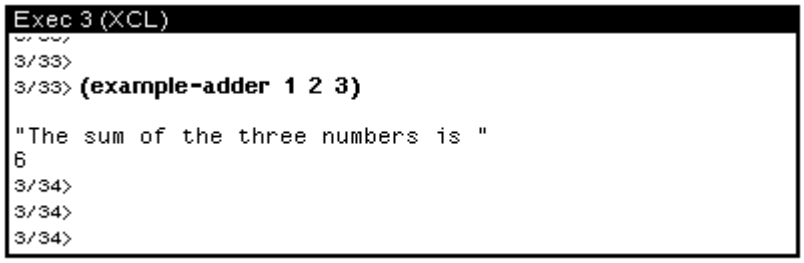


Figure 7-2.. After EXAMPLE-ADDER is defined, it can he executed

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:

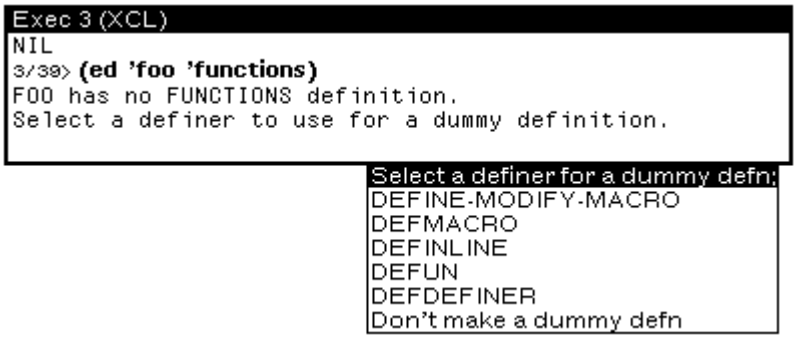


Figure 7-3 Selecting a Function Template

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.)

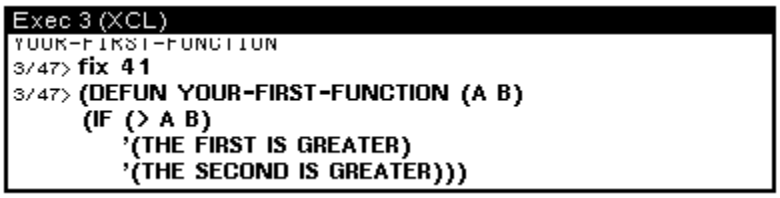


Figure 7-4. Using FIX to Edit a Fundion

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.

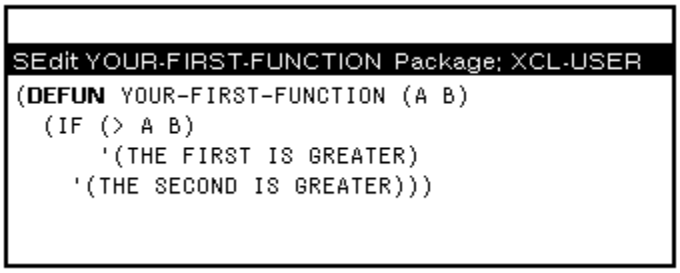


Figure 7-5. An Edit Window

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.

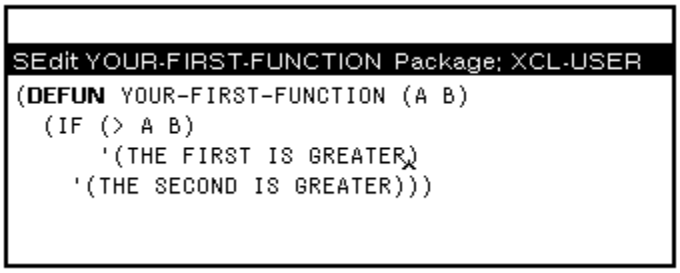


Figure 7-6. Caret Placement Prior to Changing GREATER with LARGER

Now press the DELETE key seven times, and type in LARGER. The window now looks like this:

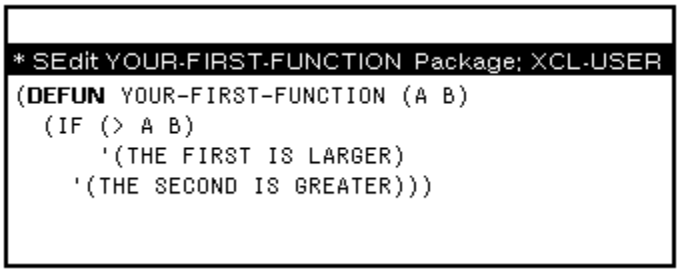


Figure 7-7. GREATER Changed to LARGER

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.

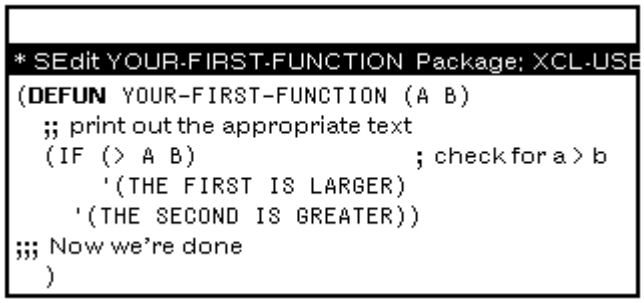


Figure 7-8. Placement of Comments

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:

```
80← 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}<LISPFILES>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.

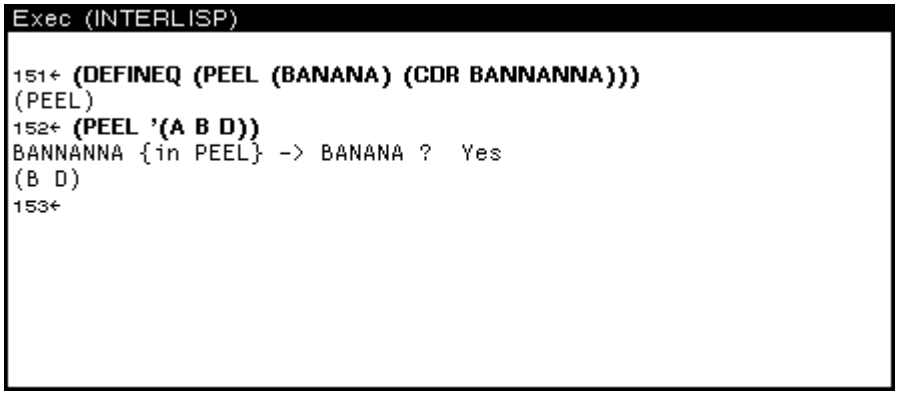


Figure 9-1. Examples of DWIM Features

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".

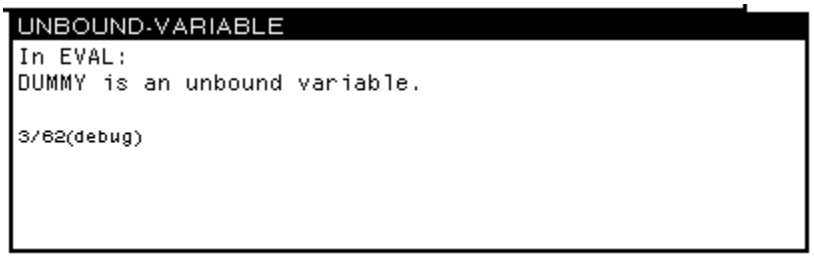


Figure 10-1. Break Window

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).

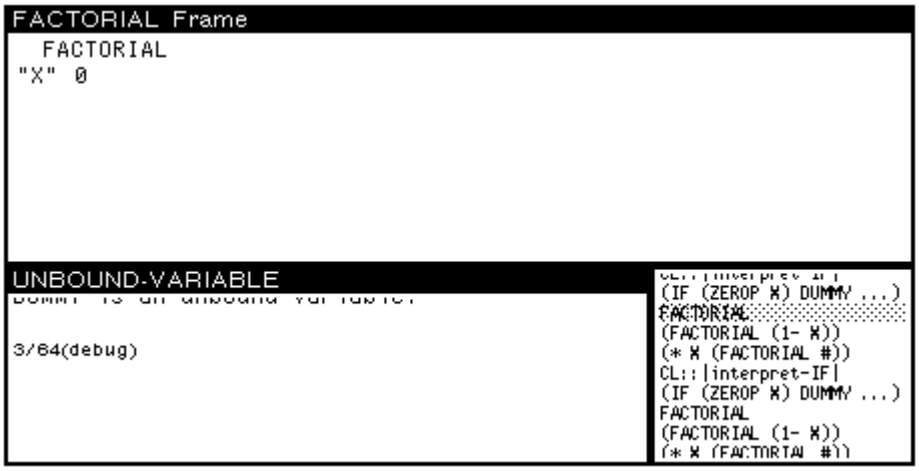


Figure 10-2. Back Trace of the System Stack

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 REVERT 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.

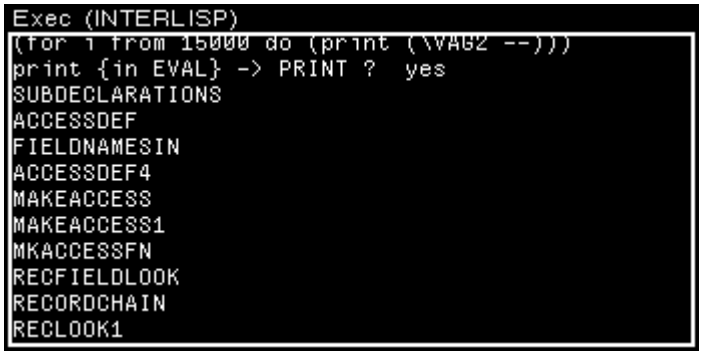


Figure 11-1. Blackened Executive Window

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.

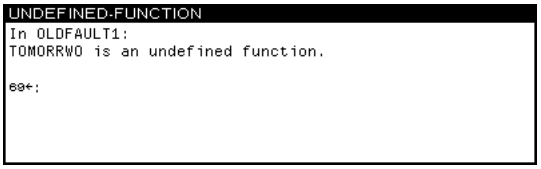


Figure 11-2. Second Window Appears (Break Window) after Typing Error Made

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.

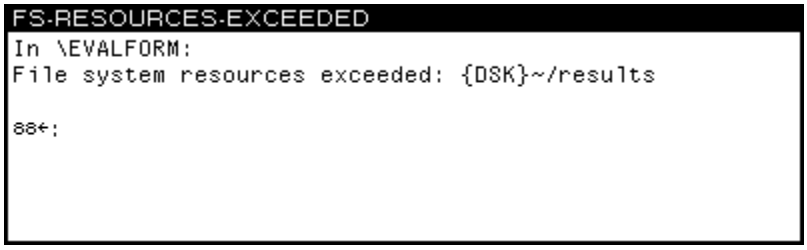


Figure 11-3. Break Window Caused by Insufficient Space in Save File

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!

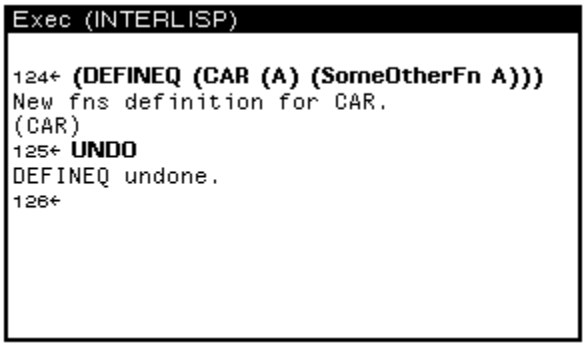


Figure 11-4. CAR redefined!

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).

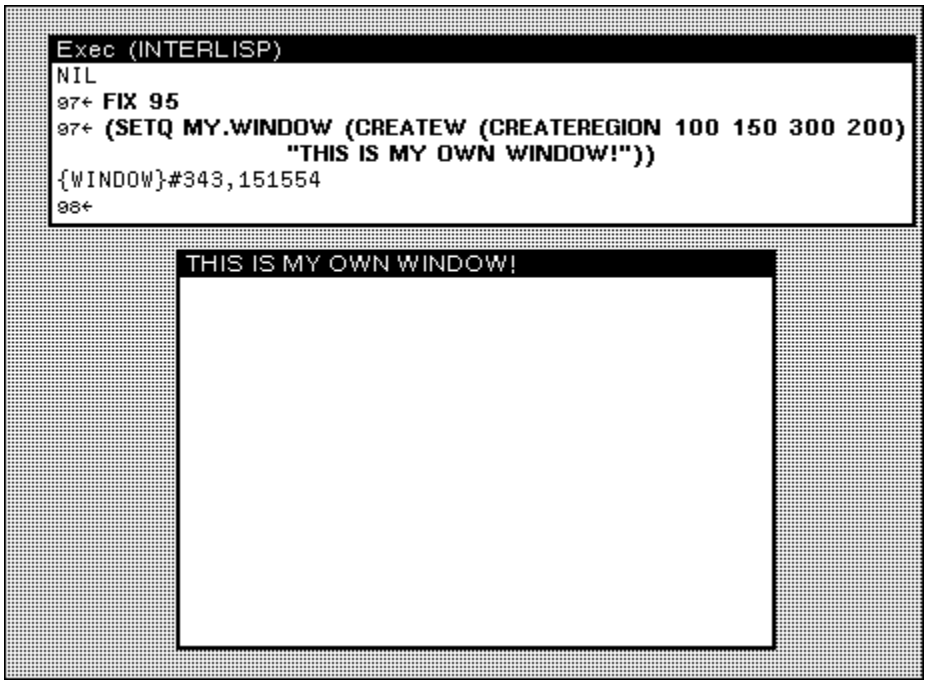


Figure 12-1. Creating a Window

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.

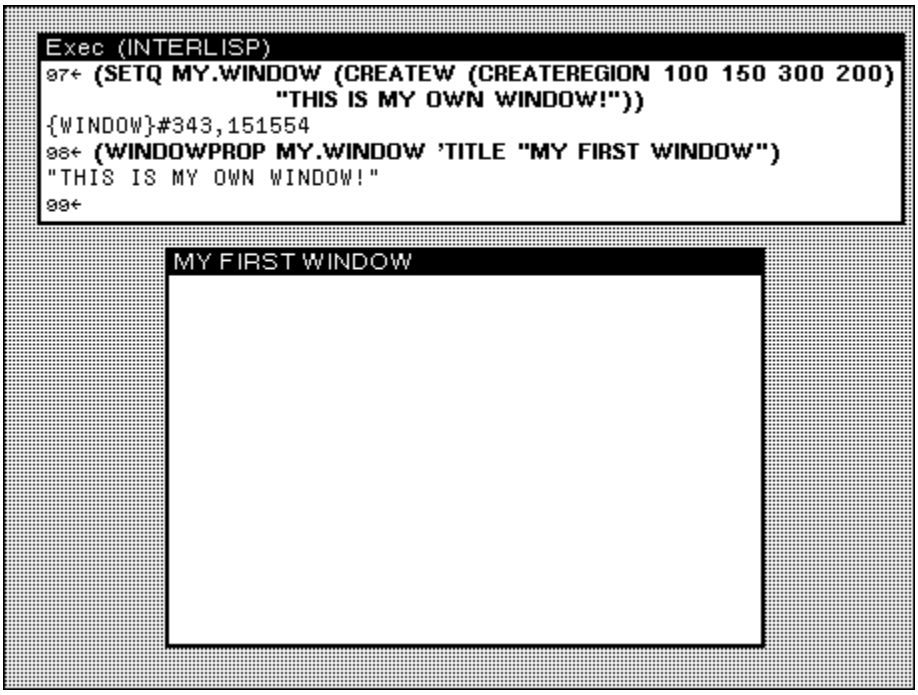


Figure 12-2. TITLE is a Window Property

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 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.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, 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 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.

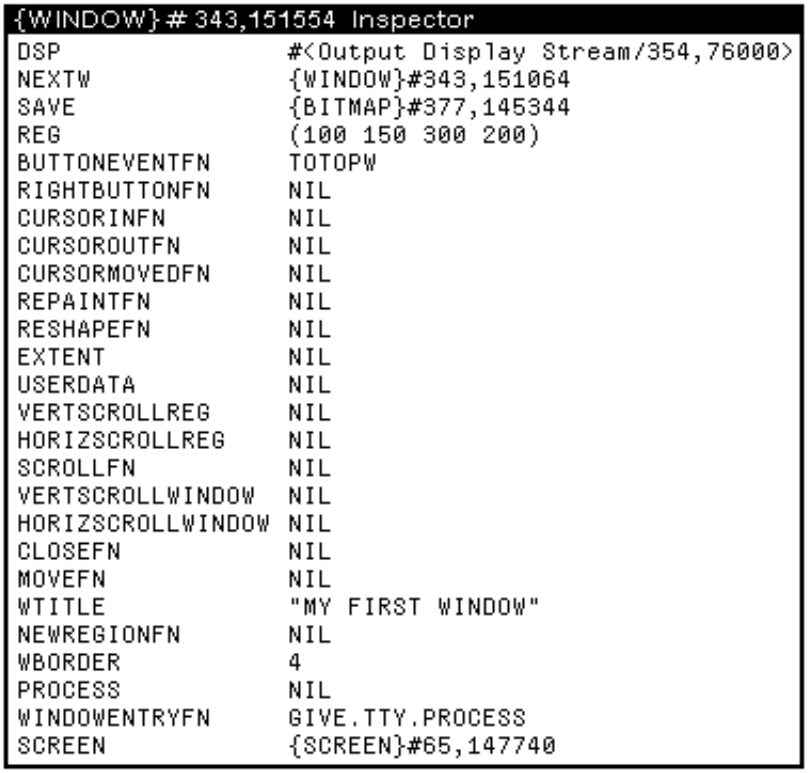


Figure 12-3. Inspecting MY.WINDOW for Mouse-Related Window Properties

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 CURS0RlNFN, 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:

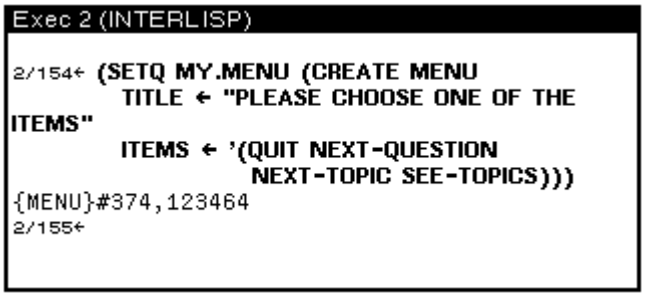


Figure 13-1. Creating a menu

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.

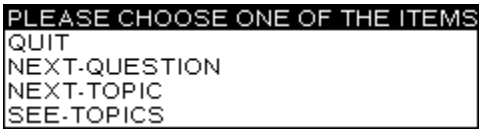


Figure 13-2. Simple MenuDisplayed with ADDMENU

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**.

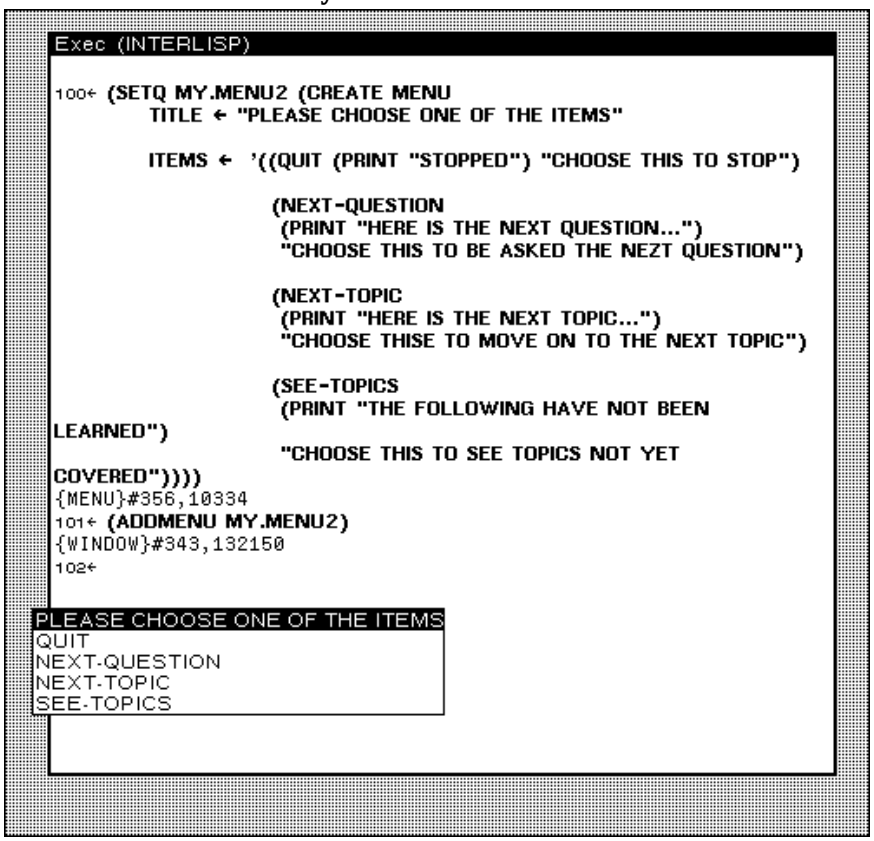


Figure 13-3. Creating a Menu to do Things, then displaying it with the function ADDMENU

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.)

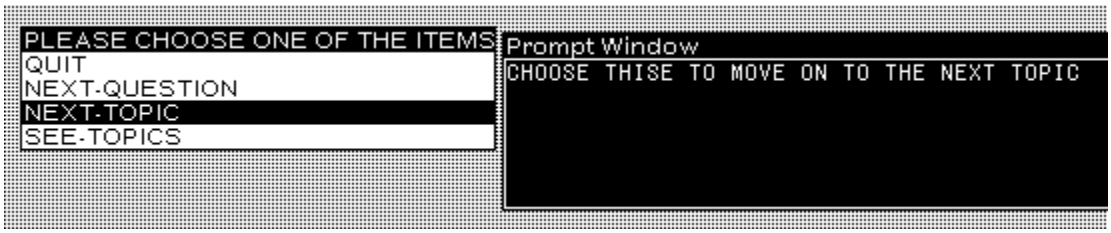


Figure 13-4. Mouse Button Held Down While Mouse Cursor SeIects NEXT.QUESTION

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.)

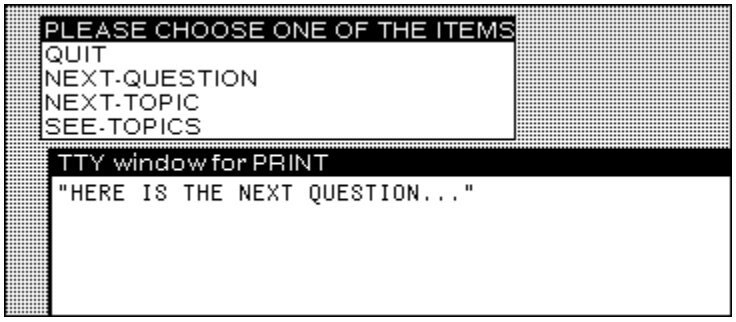


Figure 13-5. NEXT-QUESTION Selected

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.

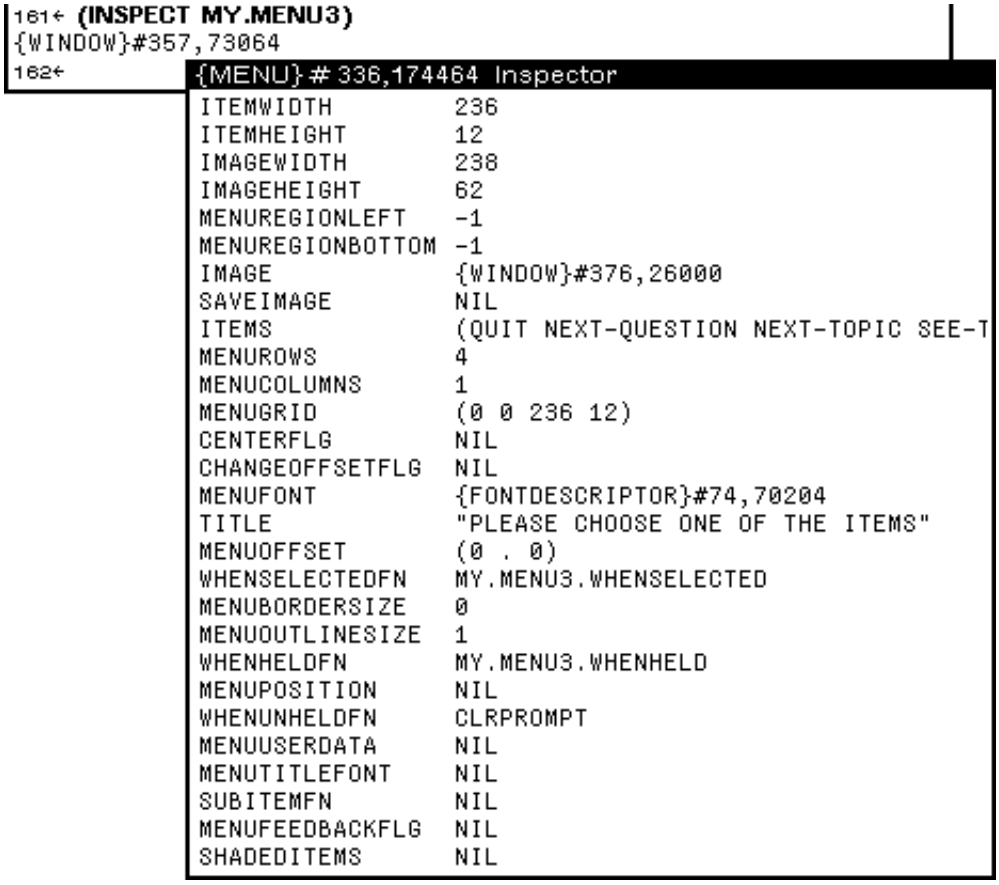


Figure 13-6. MY.MENU3 Fields

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.

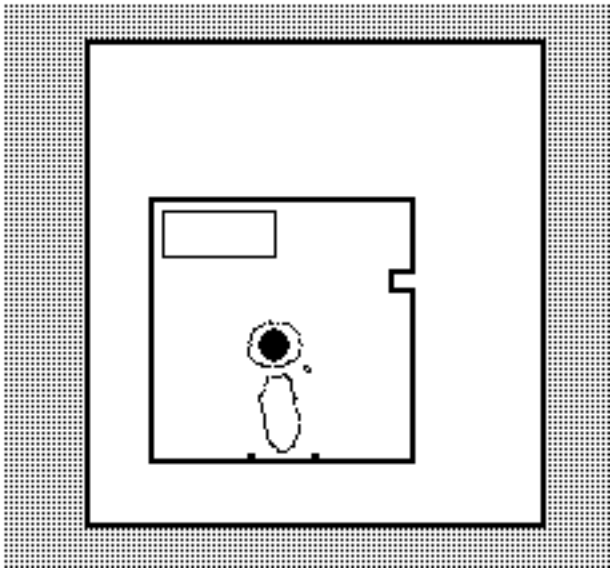


Figure 14-1. Bitmap of a Floppy

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.

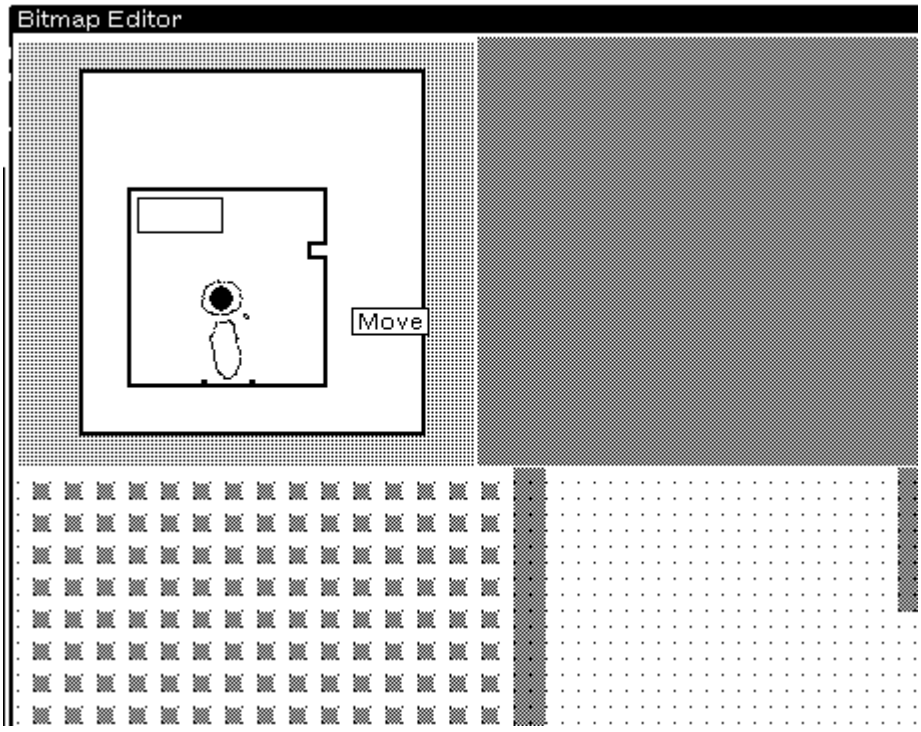


Figure 14-2. Menu with Single Item (Move)

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).

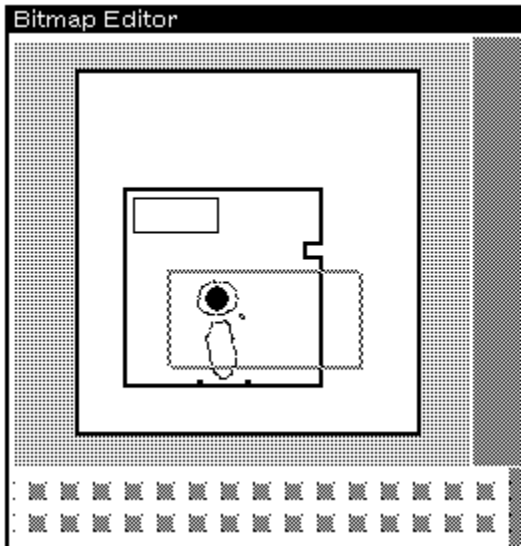


Figure 14-3. Ghost Window Awaiting Positioning

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.

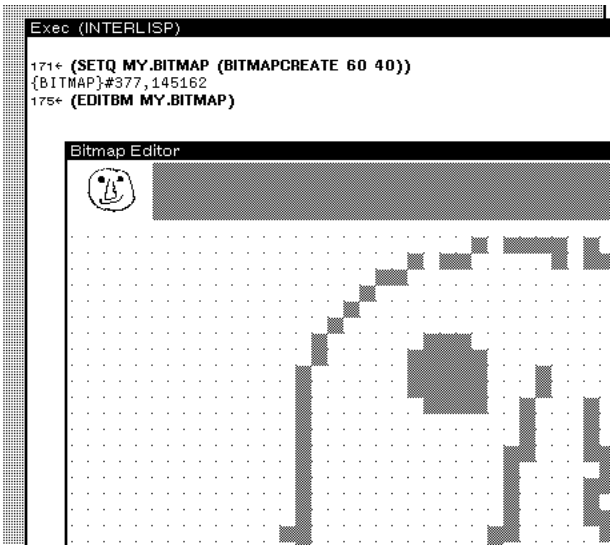


Figure 14-4. Editing a Bitmap

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).

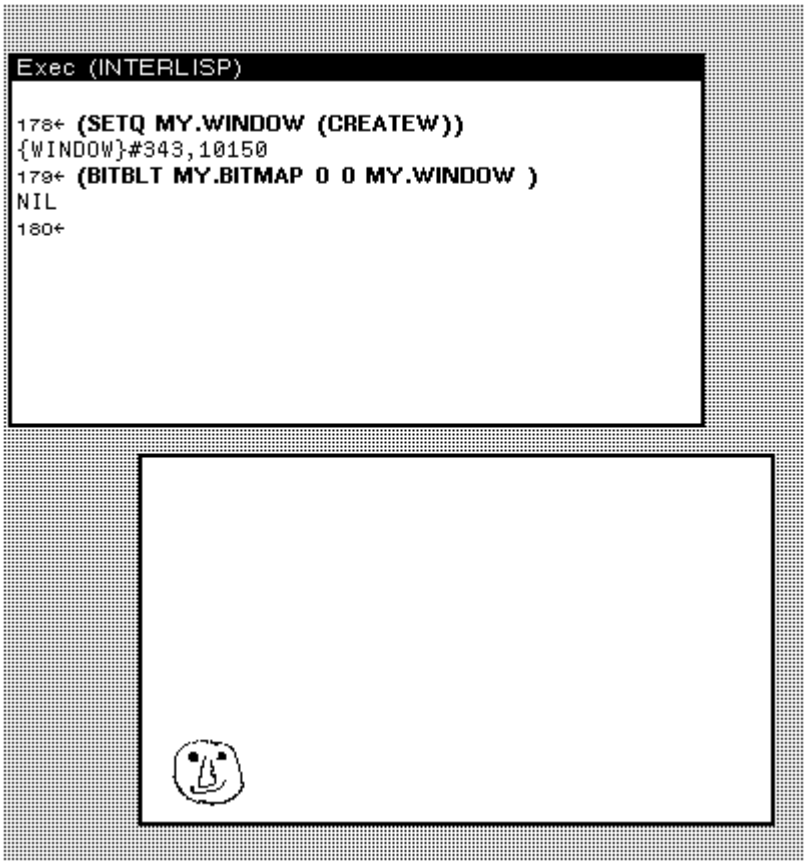


Figure 14-5. BITBLTing a Bitmap onto a Window

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:

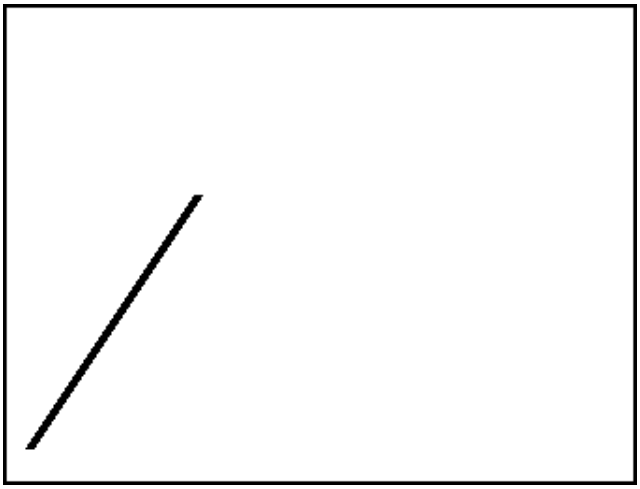


Figure 15-1. Line Drawn onto the EXAMPLE.WINDOW Displayrtream

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:

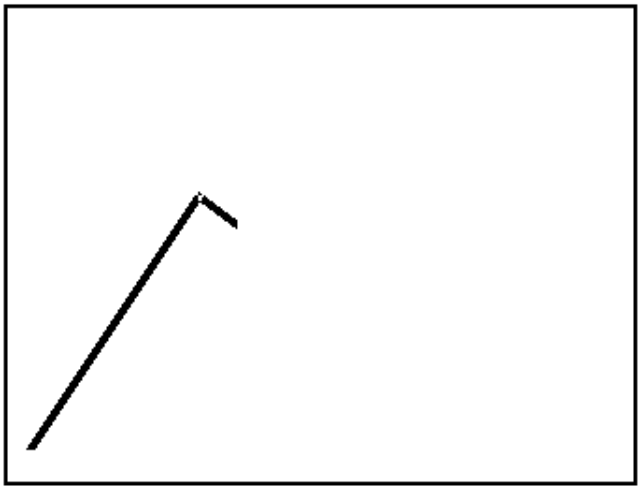


Figure 15-2. Another Line drawn onto the `EXAMPLE.WINDOW` Displaystream

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:

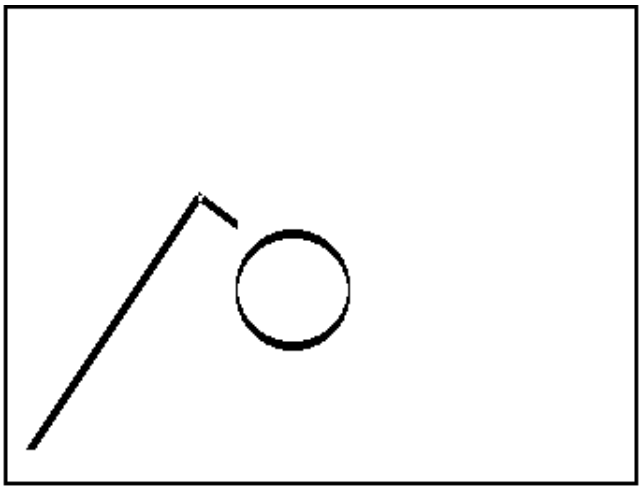


Figure 15-3. Circle Drawn onto the EXAMPLE.WlNDOW Displaystream

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:

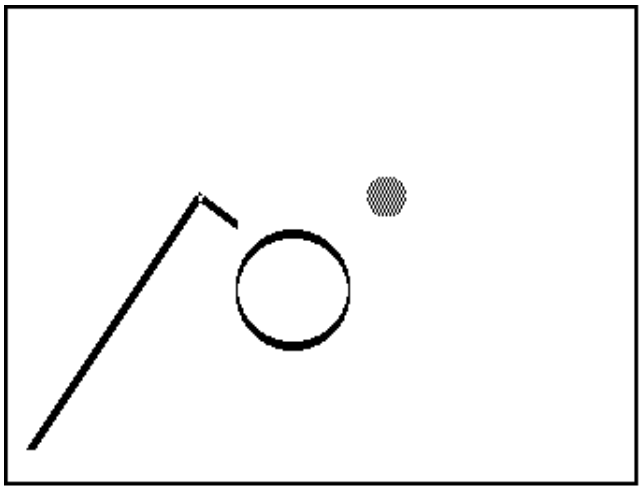


Figure 15-4. A filled circle drawn onto the displaystream

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:

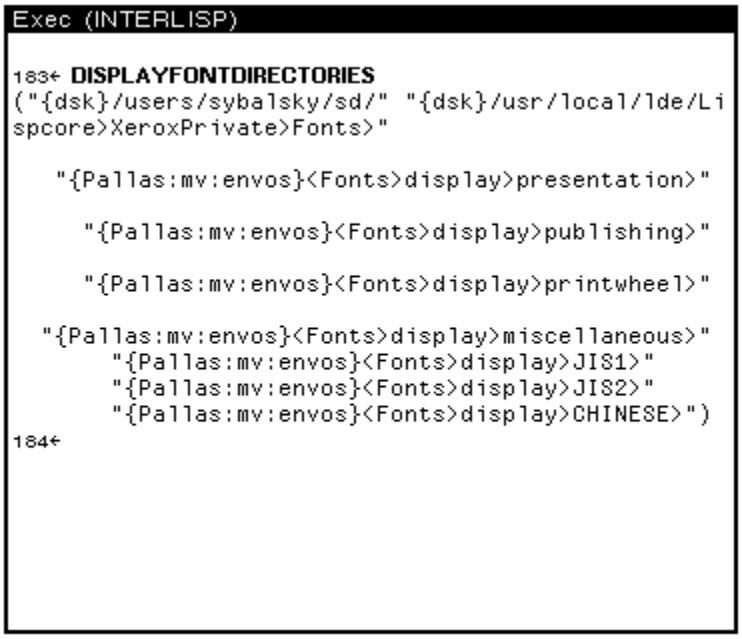


Figure 16-1. Value for the Atom DISFLAYFONTDIRECTORIES

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:

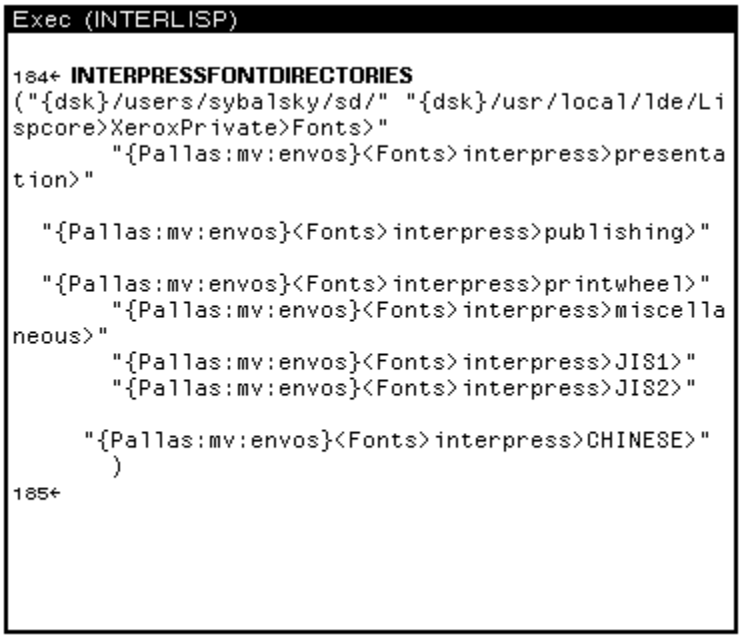


Figure 16-2. Value for Atom INTERPRESSFONTDIRECTORIES

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:

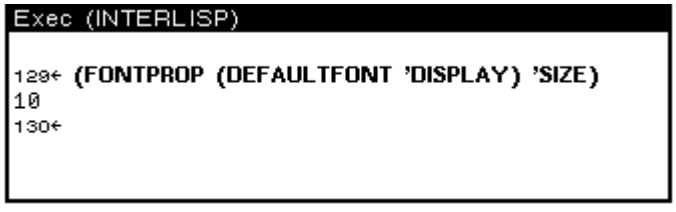


Figure 16-3. Value of Font Property SIZE of Default Font

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:

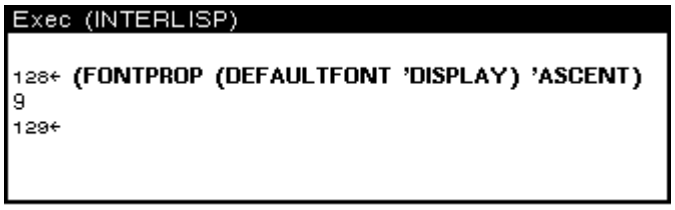


Figure 16-4. Value Font Property ASCENT of Default Font

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:

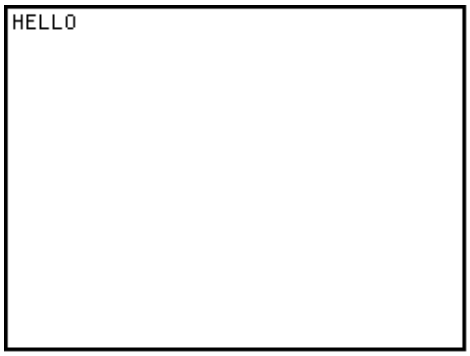
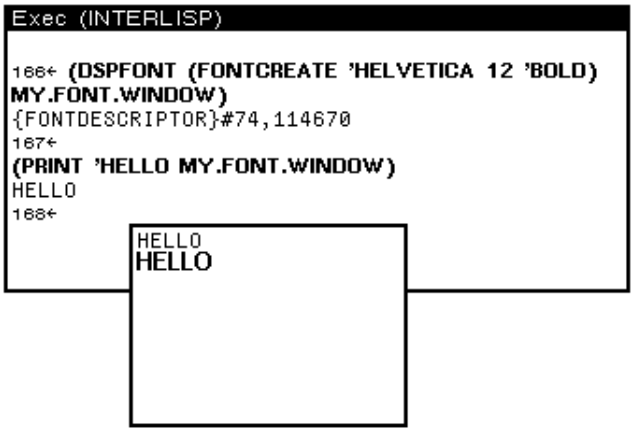


Figure 16-5. HELLO, Printed with the Default Font in MY.FONT.WINDOW

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.

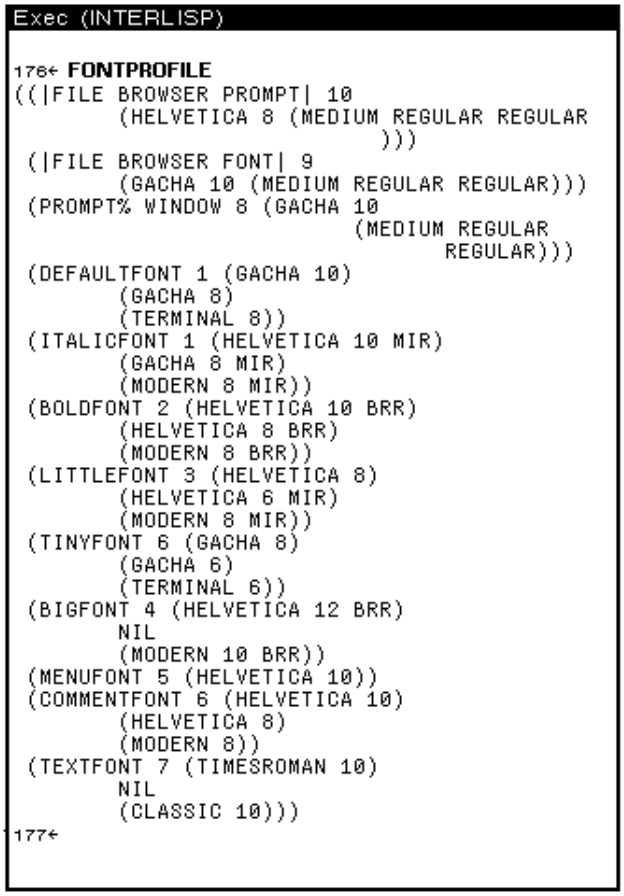


Figure 16-7. Value of the Atom FONTPROFILE

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.

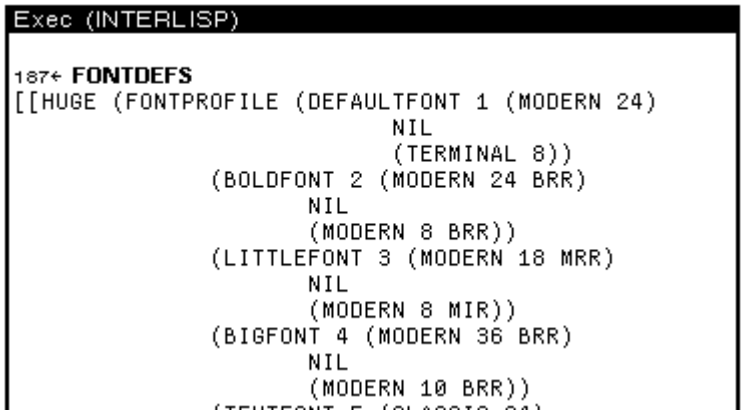


Figure 16-8. Part of Value of the Atom FONTDEFS

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.

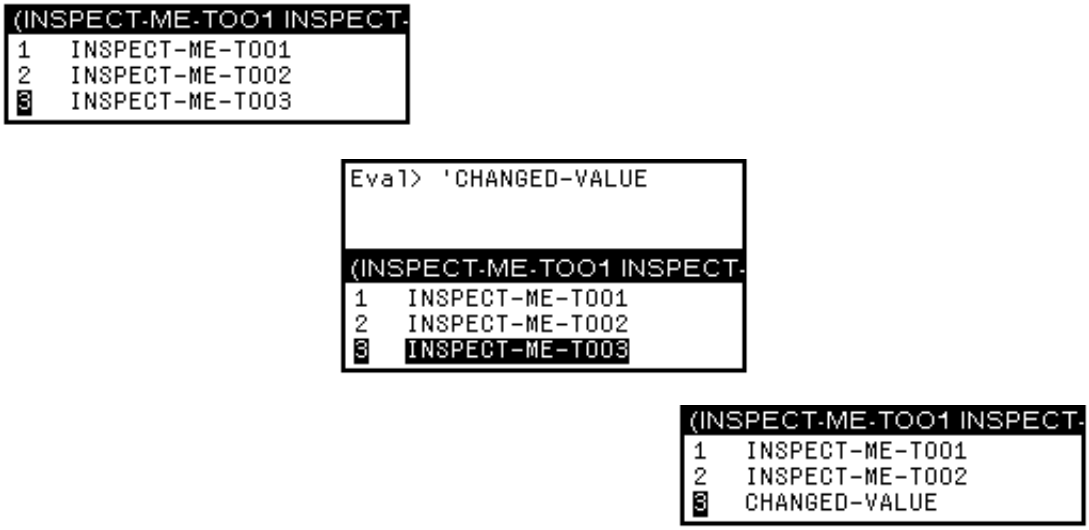


Figure 17-3. Steps Involved in Changing Value in Right Column of Inspector Window

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.

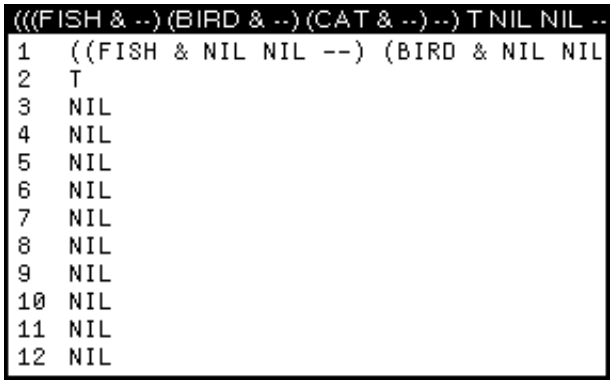


Figure 17-4. Inspector Window For ANIMAL.GRAPH, Inspected as List

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.

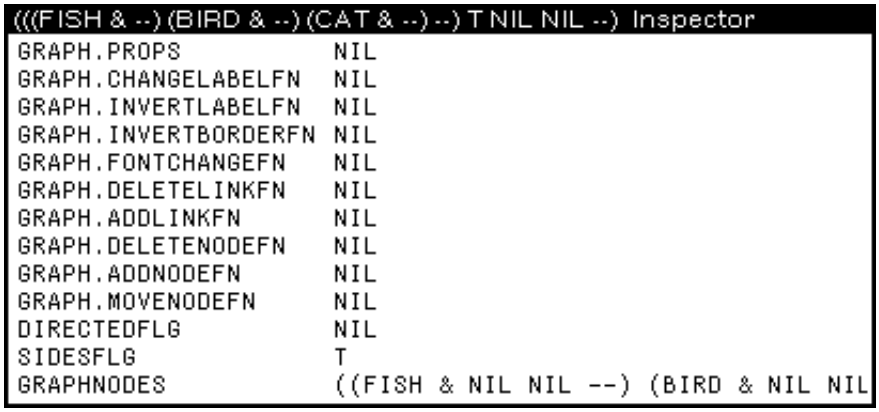


Figure 17-5. Inspector Window for ANlMAL.GRAPH, Inspected as Instance of GRAPH Record

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.

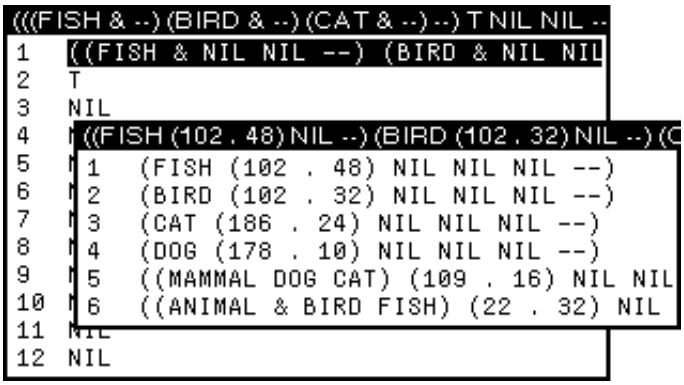


Figure 17-7. Inspector Window for ANlMAL.GRAPH and for First Element of ANIMAL.GRAPH

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.)

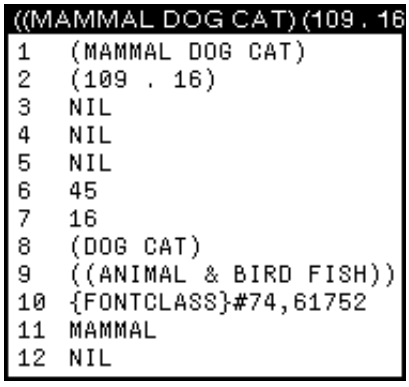


Figure 17-8. Inspector Window for Element S From Figure 17.7 That Begins ((MAMMAL DOG CAT).

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.

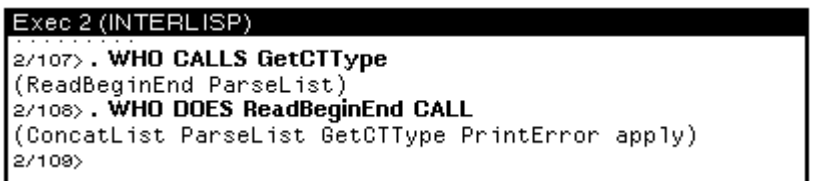


Figure 18-2. Sample Masterscope Output

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.

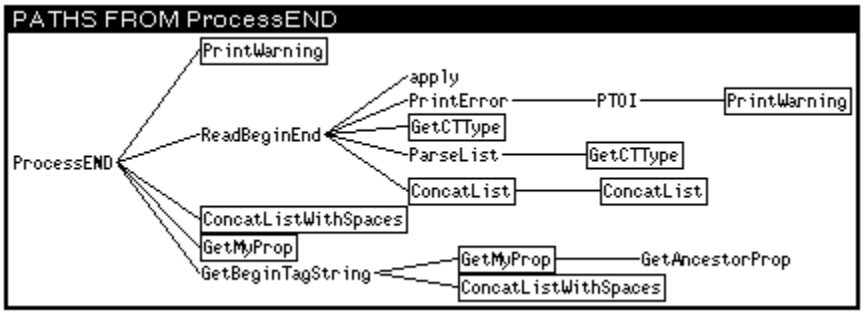


Figure 18-3. SHOW PATHS Display Example

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).

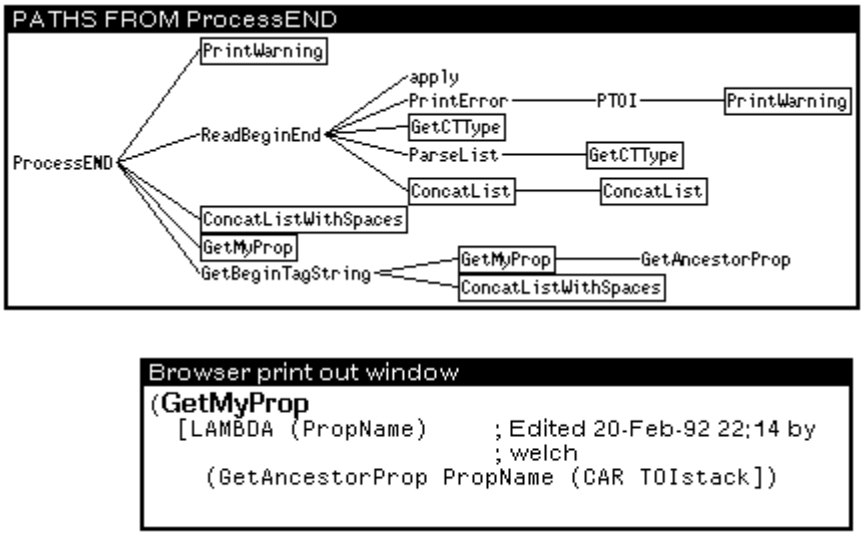
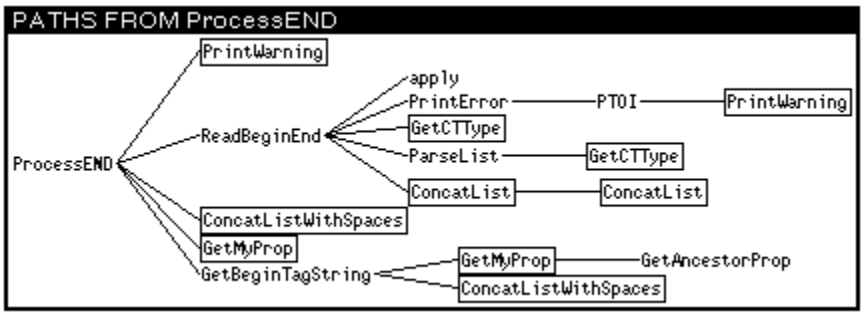


Figure 18-4. Browser Printout Example

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).



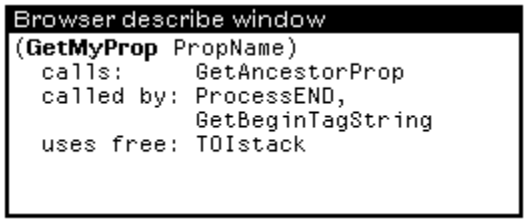


Figure 18-5. Browser Description Example

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.

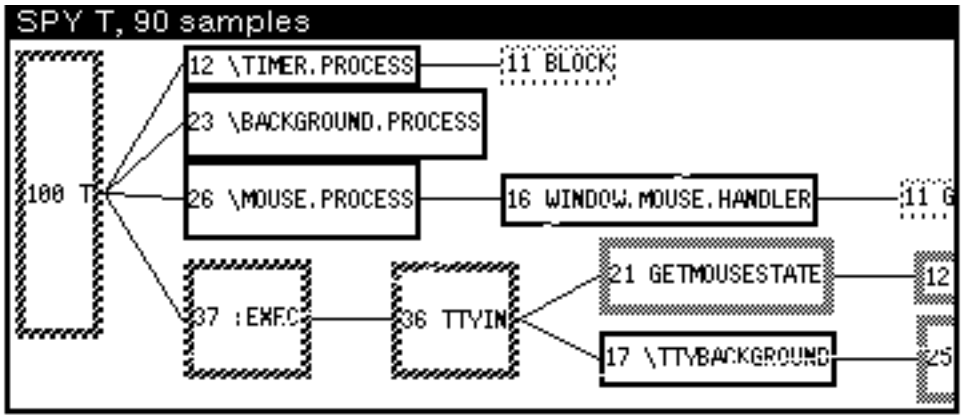


Figure 19.3. Window Produced After Running SPY

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.

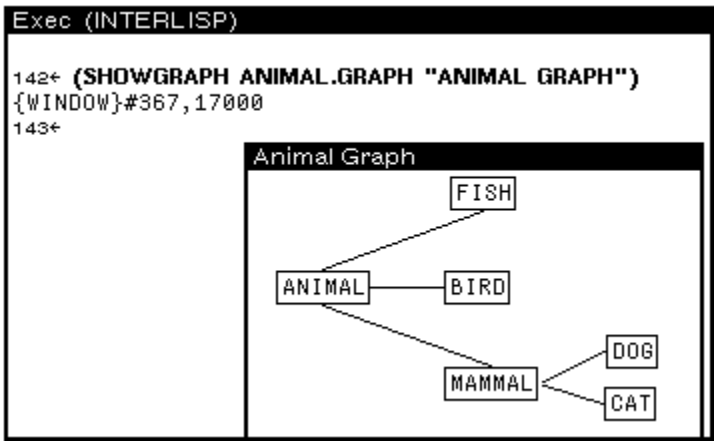


Figure 21-1. Simple Graph

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.

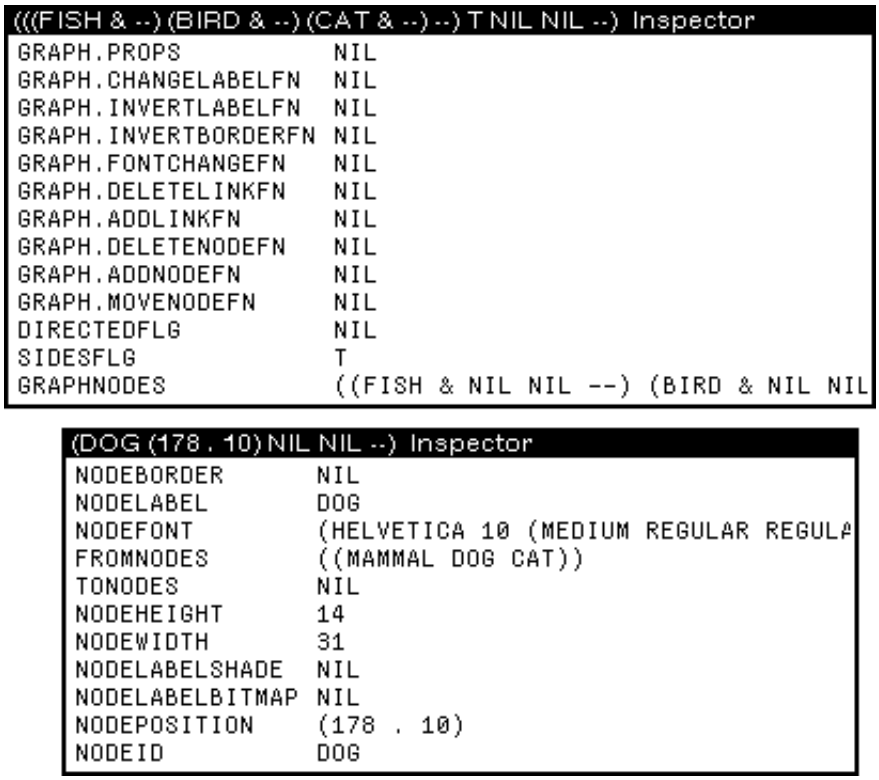


Figure 21-2. Inspecting a Graph and a Node

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)
```

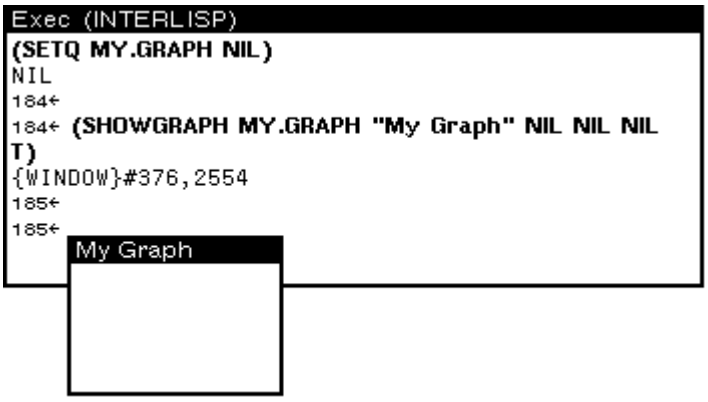


Figure 21-3. My Graph

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.

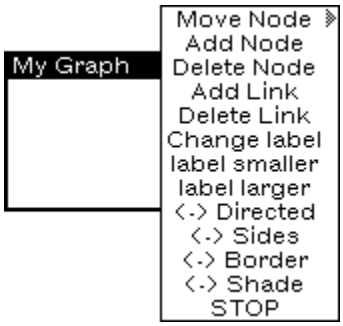


Figure 21-4. Menu of Graph Editing Operations

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.

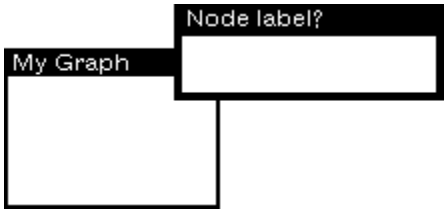


Figure 21-5. Grapher Prompts for Name of Node to add after Add Node is Chosen from Graph Editing Menu.

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.

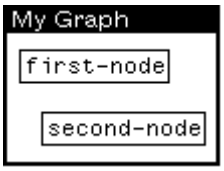


Figure 21-6. Two Nodes Added to `MY GRAPH` Using GraphEditing Menu

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.

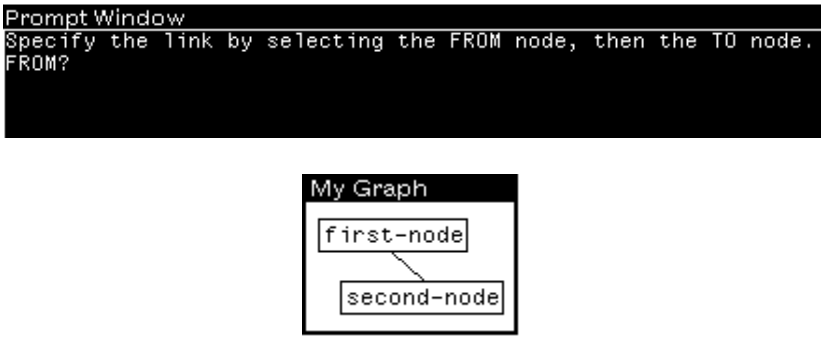


Figure 21-7. Prompt Window Requesting Selection of Two Nodes to Link, and Result

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.

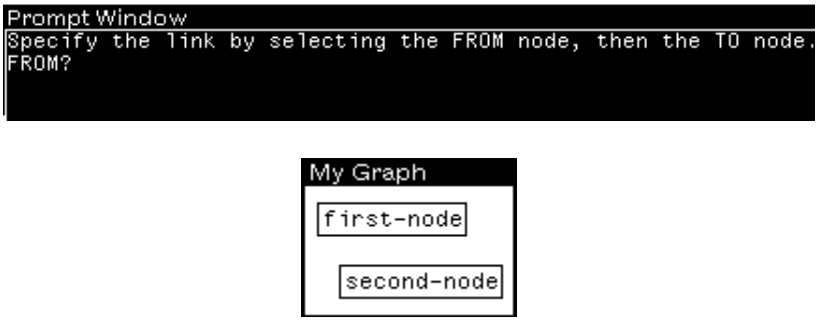


Figure 21-8. Prompt Window Requesting Selection of Link to Delete, and Result

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:

```
(RECORD 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.

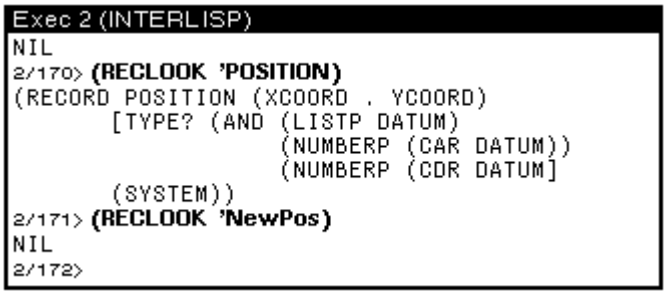


Figure 22-1. Response to RECLOCK

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.

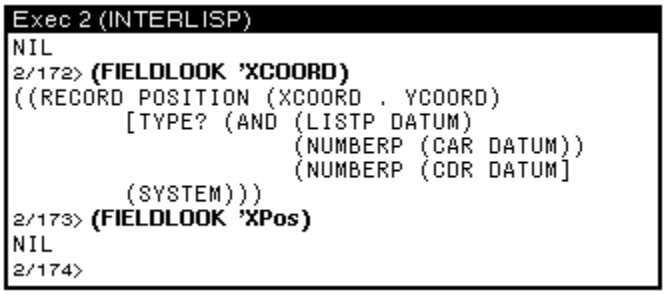


Figure 22-2. Response to FIELDLOOK

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.

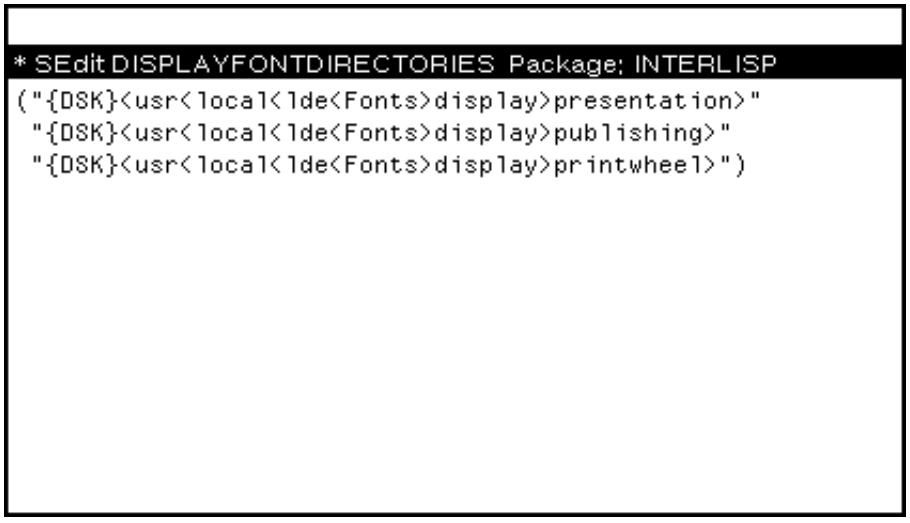


Figure 24.1. Value for the Atom DISPLAYFONTDIRECTORIES

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
- ALWAS 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:

- [redacted]E[redacted] isler. 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 ~'~

TA8~ OF CONTENtt To'.,
1

- - - -

. - - - - . - - , -

TABLE OF CONTENtr

YA8~ OF CONTENTS TOC.3

TABLE OF CONTEND yl

----- 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 GL0SSARY 1.1
1

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

1,

AøR1EFGLos~y
l

A øRl—F GLoSSARY

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.

A BRIEF GLOSSARY lJ
1

A BRIEF GLOSSARY

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

I.A A llEF GLos~Y
l

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 1.5
I

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 flløcontaining G wl'0lø Lisp environmønt: namely, lnterlis~O, evørything thø user' døfinecl or loaded into the environment, thø

1.6 A ~N:EF GL0SsARY
I

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

TYPING 5H0RTCUTS 6 1

TYPING SHORTCUTS

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

USING MENUS 71
I

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.

7.2 USING NENuS
I

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

DlR—G0R1E5

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'.\PFIL—.'.'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,R—Fc.TEDIT 'PF;IMEFRøEFOø.T—DITJ
t'Dcxl,(.LIsPFILEs.PRIM—P.;ø.PRIMEP.fiEFs.TEDIT;1

Figure 8.3. An example of the use of the function COPYFILE
To delete a file: (DEl.FIL— '(~)fi1on~)
An example of this is shown in Figure 8.4.
ø ,, OELFIL— 'L'AMPLE.TEPITJ
ø . \l.. I'PfILE;'."PRIMER?>AnPLE.T—PIT;1

FigureS.O. The function DELFlLE

To rename a file: (RENlEFILE ø(osK)oldftlønrn '(rSF)ner,r11øn~)

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'PFIL—Cv;~PRIh1—R.~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

8.1 HOW T0 us: FILES

FILE VERSION NUMBERS

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.

HOW TO USE FILES as

----- 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.

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.

THOSE WONDERFlJL WINDOWS' 103

SCROLLABLE WINDOWS

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.

10.4 THo¤a ~HOERFUL~~w51

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. —DlTUNG 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 EXAMPL—øADDER 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

EDITING AND SAVING 111
1

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 PJ—~-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)
(TN— 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

EDITING AND SAVING 115
1

USING ~E LIST STRUCTURE EDITOR

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.

EDITING AND SAVING 117
I

USING THE LIST STRUCTURE EDITOR

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.

EDITING AND SAVING 119
I

----- 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.

FLEXIBILITY AND FORGIVENESS. cLIsP AND DWIM 13 I
I

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.
(D—FIKEQ (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 "łrOMn—NT~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.

8REAKPAcKAG— 14.3
I

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

1(. 8~xpAcl:AGE

E~

BREAK MENU

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.)

WINDOWS

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.
(DEFIN—Q (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:

WlNDow5 AND REGIONS 273

WINDOWS

(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~ø‡3,1jo52‡
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 BUtrON—VENTFN

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:

27A ~N00wS ANO REGIONS

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 (RIKB—LLS))))

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 CHCio8— 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

WHAT ARE MENUS' 281

DISPLAYING MENUS

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".

JGJ WHAT AR5 MENUS?

GErn~ MENUS TO DO STUFF

NIL

```
17+(SETQ Nv.MENU2 (SR—ATE 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 QLl—STIOH .
øu'HOOSE THIS TO ~E lSKED THE NE."T QUESTION"',

iNE!~T-TOPIL

(PRINT øøHERE IS THE NE'~T TOPIL .
"C.HOOSE THIS TO KOv— OH TO THE NE'\T SueJELT" '1

(SEE-TOPICS

(PRINT "THE FOLLOYIN6 HA'\E NOT e.EEN L—ARNEO"',
*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

WHAT ARE MENUS' 283

GEHlNG MENUS TO DO STUFF

28.2.1 _ The WHENH—LDFN _ and WNENSEL—CTEDFN 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 WHENS—L—CTEDFN 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 L—CTED

(SELEcfQPiNTEENNUJSENHENHELO (ITEl.S—LECTED 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 L—ARNEDø))
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 WH—NNELDFN 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

21.1 ~YAR1".NUs?

GETTlMG MENUS TO 00 STUFF

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.l‡5lSjl

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.FCTE—l

'1flE'flELDFH NV flEPlLl3 \ørtEHHELJP
ø ENl)NHELoFH l:LFF'RCHPT

ø flENOFEEOe4l,'r.FLG NIL
Figure 28.6. The Fields of MY.MENU3

WHAT ARE MENUS' 285

----- 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

BlTMAPS 291

BlTMAPS

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.BlTMAPIø6',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.)

BlTMAPS 293

~17MAP5

98'(BITBLT KY Strap NIL NIL my ,1(10p,, FL 'IL NIL HIL Tipil' P.—PLlfi

(~=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

BITMAPS 295

----- 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~IEL— 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:

DlSPLAYSTREAMS 303
l

DRAWING ON A DISPLAYSTREAM

Figure JO.t A filled circle drawn onto the displaystream, EXAMPLE WINDOW
The syntax of FILLCIRCLE i5

(FILLCIRCL— 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.

30.4 0lSPLAY¤TREA~

r.

LOCATING AND CHANGING YOUR POSITION IN A DISPLAYSTREAM

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.

DISPLAYSTREAMS 30 5

----- 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

FONT5 311
1

WHAT MAKES UP A FONT?

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

FONTS 313

INTERPRESS FONT5 - THEIR FILES, AND HOW TO FIND THEM

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:EcTnRI—~,~
.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:—KT)
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'NTcREAT— '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!' P—F"ULTFClFlT i ,.' ','cH4 LLT;
;ø','~LHk aj
t'TEPMINk'L Sij

','BlLPF'lNT :' (H—LlETIl='n' Jo E,PP.;1
'lH—L':'—TIC" L=' BPP,i
'llJPEPtl in' ~FF)

','LITTL—FC'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 LITTL—Fi)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 LITTLEFciføIT;
'.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 FONTPROFIL—. 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.L15—J

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 D—dit 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 (D—ditLingcr 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:s—E.
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-:
—::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
1n—N—l,~ lrireFiøJ.,rningø
Din" 'pe~øbøø'c8cJ1nT;.,='ør1r1n0.
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

MAST'RS~OPE 333

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.

33.4 N~TERSCOPE
1

----- 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.

WHERE D0ES ALL THE TIME G0' SPY 341
1

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~—&-

. REPE,,TE&L'.EV~rn EJ~rn ø1 EF.—UFE

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.

34.2 WHERE 00E5 ALL TN5 ylMff G0? spY
1

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

THE INSPECTOR 321

USING THE INSPECTOR

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-hl—-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.c—LETENC~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.

THE INSPECTOR 32 j

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~REhpYfi—&.

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.

34.2 WHERE nQE¤ ALL THE TIME G0? SPY
1

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.)

1

----- 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:

FREE MENUS 361
1

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:

36.2 FREE MENUS
1

PARTS OF A FREE MENU ITEM

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
s—LEcTEDFK RIKBELLS)

TOGGL— 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)

FREE MENUS 36)

I

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 GRAPHNOD—S - 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.

THEGRAPHER 371

SAY lTWlTH GRAPHS

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.

THEGRAPHER 37.3

SAY IT WITH GRAPHS

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

37.4 THEGRApHER

MAKING A GRAPH FR0M A LIST

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:
(DEFIK—Q (~.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 Nl— 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\
(NUl18—P~P !',C4l'~ D~TUfi1:))

(i\"('~'T—h1\\j (NUMBER— (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)

RESOURCE MANAGEMENT 413

----- 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---->

SOME SPACE AND TIME CONSIDERATIONS

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.

RESOURCE MANAGEMENT 41 S

----- 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.)

SIMPLE INTERACTIONS M'ITH THE cuRsoR. A BlTMAP, AND A WINDOW 42 f
I

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 (CURSORCR—Rrt (6I~PCREAtt 1 l) 1 11

ø2.2 SINPU NVE~CYI0NS WITH Tn: CURSOR. A StTNAP. ANO A WIHIOOW

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 collød 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 (PLAYTUN— (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 UNTlic—/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~ (CR—ATEI))
```

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 (—DITl))
```

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.

u.6 SIMPLE lNTEPACT~NS WITH THE CURSOR. A SITMAP. AND A WlN00W

----- 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 R—FERENCES 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.

0THER REFERENCES THAT WILL BE USEFuL T0 You 441
I

OTHER REFERENCES THAT WILL BE USEFUL TO YOU

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:'

ill OTHER REFERE~E5 THAT WILL 55 usEFuL TO You

I

----- End Forwarded Messages -----
—End of message—

APPENDIX A. THE EXEC

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 Xerox Common Lisp, the Exec acts as the top-level loop, but in addition to **read-eval-print**, it also performs a number of other tasks, and allows a much greater range of inputs. This appendix contains information from the Lyric and Medley releases. Medley additions are indicated with revision bars in the right margin.

The Exec is based on concepts from the Interlisp Programmer's Assistant (see the *Interlisp-D Reference Manual*).

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 input, in a structure called the history list. A history list is a list of the information associated with each of the individual events that have occurred, where each event corresponds to one input. Associated with each event on the history list is the input, its values, plus other optional information such as side-effects, formatting information, etc.

The following dialogue contains illustrative examples and gives the flavor of the use of the Exec. Be sure to type these examples to an Exec whose ***PACKAGE*** is set to the **XCL-USER** package. The Exec that Lisp starts up with is set to the **XCL-USER** package. Each prompt consists of an event number and a prompt character (">").

```
12>(setq foo 5)
5
13>(setq foo 10)
10

14>undo[cr]
SETQ undone.
15>foo[cr]
5
```

This is an example of direct communication with the Exec. You have instructed the Exec to **undo** *the previous event.*

. . .

```
25>set(lst1 (a b c))
(A B C)

26>(setq lst2 '(d e f))
(D E F)

27>(mapc #'(lambda (x) (setf (get x 'myprop) t)) lst1)
(A B C)
```

The Exec accepts input both in APPLY format (the **SET***) and EVAL format (the* **SETQ***.) In event 27, the user adds a property MYPROP to the symbols* **A***,* **B***, and* **C***.*

```
28>use lst2 for lst1 in 27[cr]
```

```
NIL
```

You just instructed the Exec to go back to event number 27, substitute **LST2** *for* **LST1,** *and then re-execute the expression. You could have also used -2 instead of 27, specifying a relative address.*

.
.
.

46>**(setf my-hash-table (make-hash-table))**

```
#<Hash-Table @ 66,114034>
```

47>**(setf (gethash 'foo my-hash-table) (string 'foo))**

```
"FOO"
```

If **STRING** *were computationally expensive (which it is not), then you might be caching its value for later use.*

48>**use fie for foo in string**cr

```
"FIE"
```

You now decide you would like to redo the **SETF** *with a different value. You specify the event using "***IN STRING***" rather than* **SETF***.*

49>**?? use**cr

```
   USE FIE FOR FOO IN STRING
48>  (SETF (GETHASH 'FIE MY-HASH-TABLE)
       (STRING 'FIE))
     "FIE"
```

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.

The most common interaction with the Exec occurs at the top level or in the debugger, where you type in expressions for evaluation, and see the values printed out. In this mode, the Exec acts much like a standard Common Lisp top-level loop, except that before attempting to evaluate an input, the Exec first stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modification and/or re-execution. The Exec also notes new functions and variables to be added to its spelling lists to enable future corrections.

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 entry on the history list corresponding to the input, and prints the result. Finally the Exec displays a prompt to indicate it is 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 the variable **FOO** is the list **(A B C)**:

32>**FOO**cr
(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:

123> **(+ 1 (* 2 3))**
7
124> **(+ 1 (***cr
2 3))
7

APPLY-format input

Often, when typing at the keyboard, 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 form, the symbol is **APPLY**ed 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.

125>**UNBREAK)**

is equivalent to

125>**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 a carriage return is explicitly typed.

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, so that Exec commands are package independent.) The remainder of the line, if any, is treated as "arguments" to the command. For example,

128>**UNDO**cr
mapc undone
129>**UNDO (FOO --)**cr
foo undone

are all valid command inputs.

Multiple Execs and the Exec's Type

Multiple Execs

More than one Exec can be active at any one time. New Execs can be created by selecting the Exec menu item in the background pop-up menu. When a prompt is printed for an event in other than the

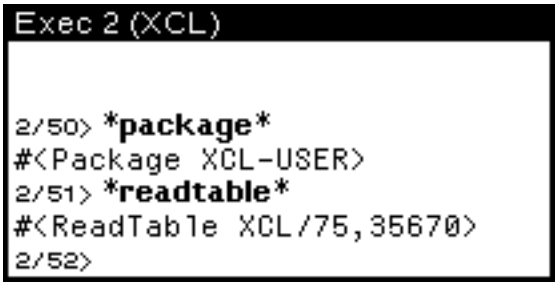
first Exec, the prompt is preceded with the Exec number; for example:

```
2/50>
```

might be a prompt in Exec 2. All Execs share the same history list, but each event records which Exec it goes with. That is, although a single global list exists, the Xerox Lisp history system maintains the separate threads of control within each Exec.

Exec type

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. Put another way, this is the Exec's mode. To allow easier setting of these modes some standard bindings for the variables have been named. The names provide the user an Exec of the Common Lisp (CL), Interlisp (IL) or Xerox Extended Common Lisp (XCL) type. An Exec's type is usually displayed in the title bar of its window in parentheses:



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.

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**.

= 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.

Note: 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* **AND** *EventAddress* , the notation *EventAddress* corresponds to all words up to the **AND** in the event specification, and *EventAddress* to all words after the **AND** in the event specification.

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** will include 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 **AND** *EventSpec* **AND . . . AND** *EventSpec*

Each of the *EventSpec* is an event specification. The lists of events are concatenated. For example, **ALL MAPC AND ALL STRING AND 32** specifies all events containing **MAPC**, all containing **STRING**, and also event **32**. Duplicate events are removed.

Exec Commands

All Exec commands are input as lines which begin with the name of the command. 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]

Similar to **REDO** except 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 correctly 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* **FOR** *OLD* **AND** ... **AND** *NEW* **FOR** *OLD* **[IN** *EventSpec* [Exec command]

Note: 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 number of expressions available, multiple **USE** expressions are generated. Thus **USE A B C D FOR E F** means substitute **A** for **E** at the same time as 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**.

Note: 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**.

? &OPTIONAL *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*.

CONN *DIRECTORY* [Exec command]

Changes default pathname to *DIRECTORY*.

DA [Exec command]

Returns current date and time.

DIR &OPTIONAL *PATHNAME* &REST *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 &REST *INPUTS* &ENVIRONMENT *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 &REST *EventSpec* [Exec command]

Edits the specified event prior to reexecuting it. If the number of characters in the **Fix**ed line is less than the variable **TTYINFIXLIMIT** then it will be edited using TTYIN, otherwise the Lisp editor is called via **EDITE**.

FORGET &REST *EventSpec* [Exec command]

Erases **UNDO** information for the specified events.

NAME *COMMAND-NAME* &OPTIONAL *ARGUMENTS* &REST *EVENT-SPEC* [Exec command]

Defines a new command, *COMMAND-NAME*, and its *ARGUMENTS*, containing the events in *EVENT-SPEC*.

NDIR &OPTIONAL *PATHNAME* &REST *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 &REST *LINE* [Exec command]

Executes *LINE* without history list processing.

UNDO &REST *EventSpec* [Exec command]

Undoes the side effects of the specified event (see below under "Undoing").

PP &OPTIONAL *NAME* &REST *TYPES* [Exec command]

Shows (prettyprinted) the definitions for *NAME* specified by *TYPES*.

SEE &REST *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, hiding comments.

SEE* &REST *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 &REST *FILES* [Exec command]

Exactly like the **TYPE** Exec command.

TYPE &REST *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,

```
312>(SQRT 2)
1.414214
313>(SQRT IL:IT)
1.189207
```

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 **-** 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 which 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 user's 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**.

Changing the Exec

(CHANGESLICE *N HISTORY —***)** [Function]

Changes the time-slice of the history list *HISTORY* to *N*. If **NIL**, *HISTORY* defaults to the top level history **LISPXHISTORY**.

Note: 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 similar to **XCL:DEFMACRO**, but defines new Exec commands. The *ARGUMENT-LIST* can have keywords, defstructure, 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 actually 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**.

Note: 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),

```
(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))
```

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**.

The new Exec now will not consider unparenthesized input with more than one argument to be in apply format. This is the same behavior as the older execs, e.g.:

```
list(1)        ; is apply format (executes after close paren is typed)
list (1)       ; is apply format (second arg is a list, no trailing args
                 given)
list '(1) 2 3 ; is NOT apply format, arguments are evaluated
list 1 2 3     ; is NOT apply format, arguments are evaluated
list 1         ; not legal input: second argument is not a list
```

Undoing

Note: This discussion only applies to undoing under the Exec, Debugger and within the UNDOABLY macro; editors handle undoing in a different fashion.

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 **UNDO**ing: one is done by the Exec, the other is available for use in a programmer's 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

Efficiency and overhead are serious considerations for the execution of a user program. Thus, the programmer 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 if you want to make a destructive operation in your own program undoable. 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-PUSH
UNDOABLY-PUSHNEW
UNDOABLY-REMF
UNDOABLY-ROTATEF
UNDOABLY-SHIFTF
UNDOABLY-DECF
UNDOABLY-INCF

UNDOABLY-SET-SYMBOL
UNDOABLY-MAKUNBOUND
UNDOABLY-FMAKUNBOUND
UNDOABLY-SETQ
XCL:UNDOABLY-SETF
UNDOABLY-PSETF
UNDOABLY-SETF-SYMBOL-FUNCTION
UNDOABLY-SETF-MACRO-FUNCTION

Note: Many destructive Common Lisp functions do not currently have undoable versions, e.g., **CL:NREVERSE**, **CL:SORT**, etc. The current list of undoable functions is saved on the association list **LISPXFNS**.

Modifying the UNDO Facility

You will usually wish to extend the **UNDO** facility after creating a form whose side effects it might be desirable to undo, for instance a file renaming function.

An undoable version of the function needs to be written. This can be done by explicitly saving previous state information away, 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 which 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 **SETF**ed. 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 **SETQ**s (and **SETF**s 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 value of **\#UNDOSAVES** is the maximum number of *UNDOFORM*s to be saved for a single event. When the count of *UNDOFORM*s 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 has been found to be a very satisfactory one. 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 have your program rewritten 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 a user seldom needs 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 the user typed-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 VALUE PROPERTY VALUE* **...)**, 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 the user 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.

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 user 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 editor for input which 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 (on 1108 and 1186)
Middle-blank key (on 1132) — 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 or *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 or *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 or *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 (on the Xerox 1108/1186/1185) or the Middle-blank key (on the Xerox 1132). 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. The notation [*CHAR*] means meta-*CHAR*. The notation $ stands for the ESCAPE/EXPAND key. 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 argument. Most of these commands are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands

[bs]	Backs up one (or n) characters.
[space]	Moves forward one (or n) characters.
[^]	Moves up one (or n) lines.
[lf]	Moves down one (or n) lines.
[(]	Moves back one (or n) words.
[)]	Moves ahead one (or n) words.
[tab]	Moves to end of line; with an argument moves to nth end of line; [$tab] goes to end of buffer.
[control-L]	Moves to start of line (or nth previous, or start of buffer).
[{] and [}]	Goes to start and end of buffer, respectively (like [$control-L] and [$tab]).
[[ ] (meta-left-bracket)	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-right-bracket)	Moves to end of current list.
[Sx]	Skips ahead to next (or nth) occurrence of character x, or rings the bell.
[Bx]	Backward search, i.e., short for [-S] or [-nS].

Buffer Modification Commands

[Zx] Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command.

[A] or [R] Repeats the last S, B, or Z command, regardless of any intervening input.

[K] Kills the character under the cursor, or n chars starting at the cursor.

[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, [<cr><cr>] will repeat the previous input (as will undo<cr> without the meta key).

[O] Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

[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.

[G] Grabs the contents of the previous line from the cursor position onward. [nG] grabs the nth previous line.

[L] Puts the current word, or n words on line, in lower case. [$L] puts the rest of the linein lower case; or if given at the end of line puts the entire line in lower case.

[U] Analogous to [L], for putting word, line, or portion of line in upper case.

[C] Capitalizes. If you give it an argument, only the first word is capitalized; the rest are just lowercased.

[control-Q] Deletes the current line. [$control-Q] deletes from the current cursor position to the end of the buffer. No other arguments are handled.

[control-W] Deletes the current word, or the previous word if sitting on a space.

Miscellaneous Commands

[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.

[N] Refreshes line. Same as control-R. [$N] refreshes the whole buffer; [nN] 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.

[control-Y] Gets an Interlisp Exec.

[$control-Y] Gets an Interlisp Exec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for Interlisp, you can do [control-L$control-Y] and give it to Lisp.

[←] 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. In Interlisp-D, the flags are all 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).

[This page intentionally left blank]

APPENDIX B. SEDIT—THE LISP EDITOR

SEdit is the Lisp structure editor. It allows you to edit 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.

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 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

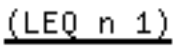
In SEdit, the mouse buttons are used as follows. The left mouse button positions the mouse cursor to point to parts of Lisp structures. The middle mouse button positions the mouse cursor to point to whole Lisp structures. Thus, selecting the Q in LEQ using the left mouse button selects that character, and sets the edit caret after the Q:

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:

At this point, any characters typed in would form a new atom between the LEQ and the n.

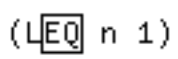
Larger structures can be selected in two ways. Use the middle mouse button to position the mouse cursor on the parenthesis of the desired list to select that list. Press the mouse button multiple times, without moving the mouse, extends the selection. Using the previous example, if the middle button were pressed twice, the list (LEQ ...) would be selected:



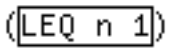
Pressing the button a third time would cause the list containing the (LEQ n 1) to be selected.

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:



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:



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: 

A gap displayed after a quote has been typed in would look like this:

with the gap marked for pending deletion, ready for typein of the object to be quoted.

16.1.5 Broken Atoms

When you are typing an atom (a symbol or a number), SEdit saves the characters you type until you finish the atom. SEdit determines that you've finished the atom when you type a character that cannot (without being escaped) belong to an atom, such as a space or open parenthesis. SEdit then tries to create an atom with these characters, just as if it were the Lisp reader. If it succeeds, the atom becomes part of the structure you're editing. However, if it fails, SEdit intercepts the reader error that would otherwise occur and instead creates a special SEdit structure called a Broken-Atom. A Broken-Atom looks and behaves in SEdit just like a normal atom, but is printed in italics to alert you to its needing correction.

SEdit has to create a Broken-Atom when the characters typed don't make a legal atom. For example, the characters "DECLARE:" cannot make 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 are not legal digits in base two, so SEdit would make a Broken-Atom that looks like *#b123*.

Broken-Atoms can be edited in SEdit just like real atoms. Whenever you finish editing a Broken-Atom, SEdit again tries to create an atom from the characters. If it succeeds, it reprints the atom in SEdit's default font, rather than in italics. You should be sure to correct any Broken-Atoms you create before exiting SEdit, since Broken-Atoms do not behave in any useful way outside SEdit.

16.1.6 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.

16.1.7 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.

16.1.8 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.

Control-Meta-O [Editor Command]

Performs a fast edit by calling ED with its CURRENT option.

16.1.9 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.

Control-Meta-X [Editor Command]

Signals the system that this edit is complete and closes the window.

Control-Meta-C [Editor Command]

Signals the system that this edit is complete, compiles the definition being editing, and closes the window.

16.1.10 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.

16.1.11 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: Control-Meta-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.

16.1.12 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 16.1.18, Options, describes the converter options .

Comment Out Selection: Control-Meta-; [Editor Command]

This command puts 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.

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 [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 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 structures contained therein.

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.

Normalize Selection: Meta-Space or Meta-Return [Editor Command]

Scrolls the current selection to the center of the window. Similarly, the Space or Return key can be used to normalize the caret.

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.

16.1.13 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.

16.1.14 Help Menu

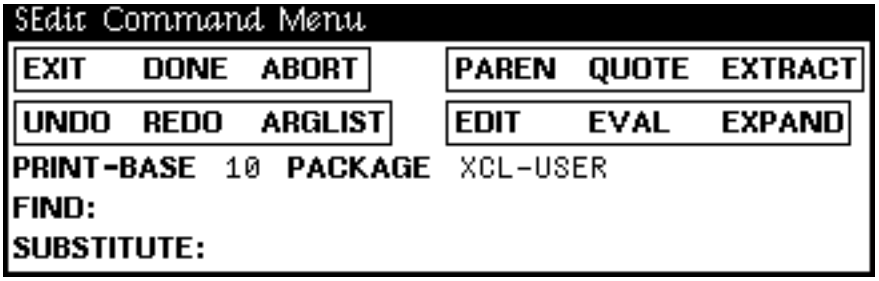
When the mouse cursor is positioned in the SEdit title bar and the middle mouse button is pressed, a Help Menu of commands pops up. The menu looks like this:

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.

16.1.15 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.

16.1.16 SEdit Programmer's Interface

The following sections describe SEdit's programmer's interface. All symbols are external in the package named "SEdit".

16.1.17 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.

16.1.18 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 struct ure, 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.

16.1.19 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, then 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 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.

edit-fn [Variable]

This function is funcalled with the selected structure and the edit options as its arguments 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:

: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.

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]

APPENDIX C. ICONW

ICONW, used to build small windows that will appear as icons on the display, is a standard input/output feature. This feature was introduced in Lyric and has been enhanced in Medley. The following description of **ICONW** should be appended to Section 28.4, Windows, of the *Interlisp-D Reference Manual*. Medley changes are indicated with revision bars in the right margin.

28.4.16 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.

28.4.16.1 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 OPERATIO N***)**

[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.

28.4.16.2 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**.

28.4.16.3 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.

28.4.16.4 Sample Icons

The LispUsers StockIcons module contains a collection of icons and their masks usable with **ICONW**, including:

- **FOLDER**, **FOLDERMASK** - a file folder
- **PAPERICON**, **PAPERICONMASK** - a sheet of paper with the top right corner turned
- **FILEDRAWER**, **FILEDRAWERMASK** - front of a file drawer
- **ENVELOPEICON**, **ENVELOPEMASK** - envelope
- **TITLED.FILEDRAWER** - TitledIcon of the filedrawer front (capacity, about three lines of 10-point text)
- **TITLED.FILEFOLDER** - TitledIcon of the file folder (capacity, about three lines of 10-point text)
- **TITLED.ENVELOPE** - TitledIcon of the envelope (capacity, one short line of 10-point text)

[This page intentionally left blank]

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:

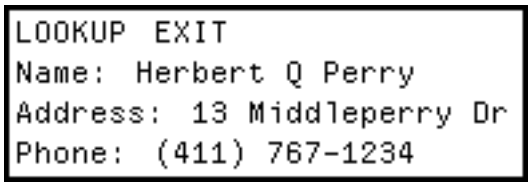
```
(               ; no PROPS list, default row format

(<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)))
```

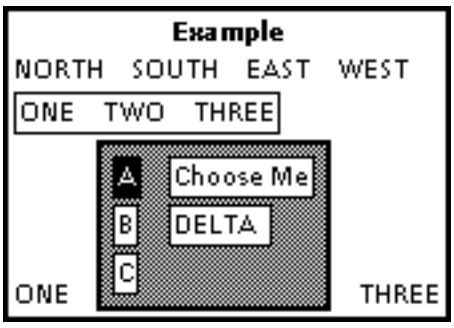
This menu has two command buttons, LOOKUP and EXIT, and three edit fields, with IDs NAME, PHONE, and ADDRESS. The Edit items are initialized to the empty string, as in this example they need no other initial value. The user could select the Name: prompt, type a person's name, and then press the LOOKUP 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:



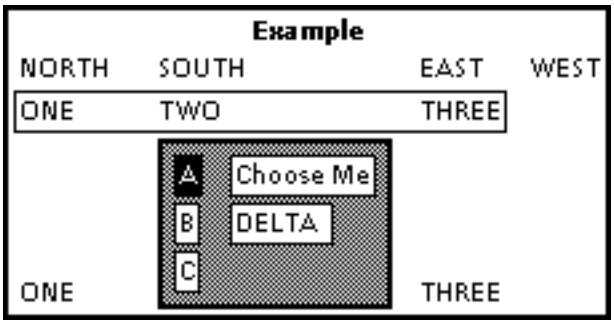
Here is a more complicated example:

```
((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))
```

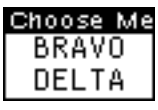
The three items, labelled A, B, and C are all declared as NWAY items, and are also specified to belong to the same NWAY Collection, Col1. This is how a number of NWAY items are collected together. The property NWAYPROPS (DESELECT T) on the first NWAY item specifies that the Col1 Collection is to have 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.

APPENDIX E. ERROR SYSTEM

This appendix replaces Chapter 24, Error System, of *Common Lisp Implementation Notes,* Lyric Release*,* which replaced most of Chapter 24, Errors, of *Common Lisp, the Language*. Text shown with ~~StrikeThru~~ is that text from the Lyric release that no longer applies in Medley. Enhancements added in Medley are indicated with revision bars in the right margin.

The XCL error system has been updated to reflect the current ANSI Common Lisp error system proposal. This version seems to be gaining wide use in other Common Lisp implementations, so no further major changes are anticipated.

The Common Lisp error system is based on proposal number 18 for the Common Lisp error system. Deviations from this proposal are noted. Since the Common Lisp error system has not yet been standardized, this system may change in future releases to accommodate the final version of the Common Lisp error system.

If you have access to the ARPANet, a copy of this proposal may be retrieved from MIT-AI.ARPA as the file "COMMON;COND18 TXT".

All symbols described in the error system proposal that are not already in the "LISP" package are exported from the "CONDITIONS" package. In addition, the "XEROX-COMMON-LISP" package exports these symbols, so you can make them available either by using "XCL" or using "CONDITIONS", whichever is appropriate to your application. The distinction is made so that XCL extensions of the Common Lisp error system will be clear. All unqualified symbols are assumed to be in the "LISP" package.

Summary of Error System Changes

The semantics of HANDLER-BIND where multiple bindings are set up or mutiple condition types are being handled are slightly different. Old code that used this will probably not behave as expected.

HANDLER-BIND and HANDLER-CASE (a.k.a. CONDITION-CASE) now always take a typespec instead of a list of condition types to indicate the conditions to be handled. Old code that uses this will only handle the first condition type in the list. The function, CONDITIONS::CONVERT-HANDLER-CASE is provided to aid in converting old code. It may be used as a mutation function in SEdit.

HANDLER-CASE now supports a :NO-ERROR option that is executed if none of the other clauses are taken. This is handy for writing code that depends on the normal completion of some operation, for example, creating auxilliary files if a particular stream is successfully opened.

SERIOUS-CONDITION no longer forces entry to the debugger. The function used to signal the condition now determines what happens if the condition is not handled. This means that SERIOUS-CONDITION has no more interesting properties and is likely to be removed in the final version of the error standard.

Several new condition types have been defined. Others have moved in the hierarchy. For example, ILLEGAL-GO is now a subtype of PROGRAM-ERROR.

No standard condition type has a default handler.

The standard debugger entry point is now called INVOKE-DEBUGGER instead of DEBUG.

The syntax of DEFINE-CONDITION has been changed to make it more like CLOS' DEFCLASS. The function CONDITIONS::CONVERT-OLD-DEFINE-CONDITION is provided to aid in converting old code. It may be used as a mutation function in SEdit.

Several DEFINE-CONDITION options have been merged, while others have been removed. In particular, there are no more "instant variables."

PROCEED-CASE has been replaced by RESTART-CASE. The semantics of restarts have been cleaned up and several new features added. Related functions, such as COMPUTE-PROCEED-CASES, have been renamed appropriately.

INVOKE-PROCEED-CASE has been renamed to INVOKE-RESTART.

DEFINE-PROCEED-FUNCTION has been removed, although XCL will continue to support it for compatibility.

The arguments to a restart's report function are different. Old code that used something other than a string for the report method will not work correctly.

A distinction is now made between invoking a restart interactively and simply invoking one. To this end, there is the function INVOKE-RESTART-INTERACTIVELY and the :INTERACTIVE option to RESTART-CASE.

RESTART-BIND, in analogy to HANDLER-BIND, has been added.

A new variable, *BREAK-ON-SIGNALS* exists to aid in debugging. It is a generalization of *BREAK-ON-WARNINGS*. The latter has been retained for compatibility.

The proceed function PROCEED has been changed to CONTINUE.

Old compiled code will continue to work except in the following cases, some of which have been mentioned above:

A proceed case's report function was not a simple string. Such code can cause stack overflow trying to report the condition (*STANDARD-OUTPUT* ends up being bound to NIL). Such code should be rewritten.

A handler binding is made to a list of condition types. Only the first type in the list will be handled.

Multiple handler bindings were created by the same HANDLER-BIND or HANDLER-CASE. Such code will work as expected, but if recompiled in Medley, will not. To get the effect of the current semantics, you must use nested HANDLER-BINDs.

Under the new error system, `use-value` and `store-value` no longer prompt for a value.

Introduction to Error System Terminology

condition A *condition* is a kind of object which is created when an exceptional situation arises in order to represent the relevant features of that situation.

signal, handlers Once a condition is created, it is common to *signal* it. When a condition is signaled, a set of *handlers* are tried in some pre-defined order until one decides to *handle* the condition or until no more handlers are found. A condition is said to have been handled if a handler performs a non-local transfer of control to exit the signalling process.

restart Although such transfers of control may be done directly using traditional Lisp mechanisms such as `catch` and `throw`, `block` and `return`, or `tagbody` and `go`, the condition system also provides a more structured way to *restart* a computation. Among other things, the use of these structured primitives for restarting allows a better and more integrated relationship between the user program and the interactive debugger.

~~*serious conditions*~~ ~~It is not necessary that all conditions be handled. Some conditions are trivial enough that a failure to handle them may be disregarded. Others, which we will call *serious conditions* must be handled in order to assure correct program behavior. If a serious condition is signalled but no handler is found, the debugger will be entered so that the user may interactively specify how to proceed.~~

errors conditions which result from incorrect programs or data are called *errors*. Not all conditions are errors, however. Storage conditions are examples of conditions that are not errors. For example, the control stack may legitimately overflow without a program being in error. Even though a stack overflow is not necessarily a program error, it is serious enough to warrant entry to the debugger if the condition goes unhandled.

Some types of conditions are predefined by the system. All types of conditions are subtypes of `conditions:condition`. That is,

```
(typep c 'conditions:condition)
```

is true if `c` is a condition.

creating conditions

The only standard way to define a new condition type is `conditions:define-condition`. The only standard way to instantiate a condition is `conditions:make-condition`.

When a condition object is created, the most common operation to be performed upon it is to *signal* it (although there may be applications in which this does not happen, or does not happen immediately).

When a condition is signaled, the system tries to locate the most appropriate handler for the condition and invoke that handler. Handlers are located according to the following rules:

bound

- Check for locally defined (ie, *bound*) handlers.
- If no appropriate bound handler is found, check first for the default handler of the signaled type and then of each of its superiors.

decline

If an appropriate handler is found, the handler may *decline* by simply returning without performing a non-local transfer of control. In such cases, the search for an appropriate handler is picked up where it left off, as if the called handler had never been present. When a handler is running, the "handler binding stack" is popped back to just below the binding that caused that handler to be invoked. This is done to avoid infinite recursion in the case that a handler also signals a condition.

`conditions:handler-bind`

When a condition is signaled, handlers are searched for in the dynamic environment of the signaller. Handlers can be established within a dynamic context by use of `conditions:handler-bind` and other forms based on it.

handler

A *handler* is a function of one argument, the condition to be handled. The handler may inspect the object (using primitives described in another section) to be sure it is interested in handling the condition. After inspecting the condition, the handler must take one of the following actions:

- It may decline to handle the condition by simply returning. When this happens, any returned values are ignored and the effect on the signaling process is the same as if the handler had not run. The next handler in line will be tried, or if no such handler exists, the default action for the given condition will be taken. A default handler may also decline, in which case the condition will go unhandled. What happens then depends on which function was used to signal the condition (`xcl:signal`, `error`, `cerror`, `warn`).
- It may perform some non-local transfer of control using `go`, `return`, `throw`, `abort`, or `conditions:invoke-restart`.
- It may signal another condition.
- It may invoke the debugger.

`conditions:restart-case` When a condition is signalled, a facility is available for use by handlers to transfer control to an outer dynamic contour of the program. The form which creates contours that may be returned to is `conditions:restart-case`. Each contour is set up by a `conditions:restart-case` clause, and is called a *restart*. The function that transfers control to a restart is `conditions:invoke-restart`.

~~*proceed function* Also, control may be transferred along with parameters to a named `xcl:proceed-case` clause by invoking a *proceed function* of that name.~~

~~*Proceed functions* are created with the macro `xcl:define-proceed-function`.~~

restart type A restart with a particular name is sometimes called a *restart type*.

report In some cases, it may be useful to *report* a condition or a restart to a user or a log file of some sort. When the printer is invoked on a condition or proceed case and `*print-escape*` is `nil`, the report function for that object is invoked. In particular, this means that an expression like

```
(princ condition)
```

will invoke `condition`'s report function. Because of this, no special function is provided for invoking the report function of a condition or a restart.

Program Interface to the Condition System

Defining and Creating Conditions

conditions:define-condition *name* (*parent-type*) [({*slot*}*) {*option*}*]

[Macro]

Defines a new condition type with the given *name*, making it a subtype of the given *parent-type*.

Except as otherwise noted, the arguments are not evaluated.

The valid *option*s are:

`(:documentation` *doc-string*`)`

doc-string should be a string which describes the purpose of the condition type or `NIL`. If this option is omitted, `NIL` is assumed. `(documentation` *name* `'type)` will retrieve this information.

`(:conc-name` *symbol-or-string*`)`

As in `defstruct`, this sets up automatic prefixing of the names of slot accessors. Also as in `defstruct` if no prefix is specified the default behavior for automatic prefixing is to use the name of the new type followed by a hyphen interned in the

package which is current at the time that the `conditions:define-condition` is processed.

~~`:report-function` *expression*~~

~~*expression* should be a suitable argument to the `function` special form, e.g., a symbol or a lambda expression. It designates a function of two arguments, a condition and a stream, which prints the condition to the stream when `*print-escape*` is `nil`.~~

~~The `:report-function` describes the condition in a human-sensible form. This item is somewhat different than a structure's `:print-function` in that it is only used if `*print-escape*` is `nil`.~~

`(:report` *exp*`)`

This option specifies the report function for this condition type. Report function are inherited, so if a particular condition type does not have one, the report function of its parent will be used.

If *exp* is a string, it is a shorthand for

```
(:report (lambda (condition stream)
        (declare (ignore conditions))
         (princ exp stream)))
```

If *exp* is not a string, `(function` *exp*`)` will be evaluated in the current lexical environment. This should return a function of two arguments, a condition and a stream. It will be called when a condition of this type is to be printed and `*print-escape*` is `nil`. The report function will be called with the condition to be reported and the stream to which the report is to be made.

~~`:handler-function` *expression*~~

~~*expression* should be a suitable argument to the `function` special form. It designates a function of one argument, a condition, which may handle that condition if no dynamically-bound handler did.~~

`(:handle` *exp*`)`

This option specifies a default handler for conditions of this type. `(function` *exp*`)` will be evaluated in the current lexical context. This should result in a function of one argument, a condition, to be used as the default handler for this condition type.

Each *slot* is a `defstruct` *slot-description*. In addition to those specified, the slots of the *parent-type* are also available. No slot-options are allowed, only an optional default-value expression. Condition objects are immutable, i.e., all of their slots are automatically declared to be `:read-only`.

`conditions:make-condition` will accept keywords with the same name as any of the slots, and will initialize the corresponding slots in conditions it creates.

Accessors are created according to the same rules as used by `defstruct`. For example:

```
(conditions:define-condition bad-food-color (food-lossage)
(food color)
      (:report (lambda (c s) (format s "The food ~A was ~A"
                        (bad-food-color-food c) (bad-food-
color-color c)))))
```

defines a condition of type `bad-food-color` which inherits from the `food-lossage` condition type. The new type has slots `food` and `color` so that `conditions:make-condition` will accept `:food` and `:color` keywords and accessors `bad-food-color-food` and `bad-food-color-color` will apply to objects of this type.

The report function for a condition will be implicitly called any time a condition is printed with `*print-escape*` being `nil`. Hence,

```
(princ condition)
```

is a way to invoke the condition's report function.

Here are some examples of defining condition types. This form defines a condition called `machine-error` which inherits from `error`:

```
(conditions:define-condition machine-error (error) (machine-
name)
      (:report (lambda (c s) (format s
                  "There is a problem with ~A."
                  (machine-error-machine-name c))))
   )
```

The following defines a new error condition (a subtype of `machine-error`) for use when machines are not available:

```
(conditions:define-condition machine-not-available-error
                     (machine-error) (machine-name)
    (:report (lambda (c s) (format s
               "The machine ~A is not available."
               (machine-error-machine-name c))))
   )
```

The following defines a still more specific condition, built upon `machine-not-available-error`, which provides a default for `machine-name` but which does not provide any new slots:

```
(conditions:define-condition
          my-favorite-machine-not-available-error
          (machine-not-available-error)
   ((machine-name "Tesuji:AISDev")))
```

This gives the `machine-name` slot a default initialization. Since no `:report` clause was given, the information supplied in the definition of `machine-not-available-error` will be used if a condition of this type is printed while `*print-escape*` is `nil`.

xcl:condition-reporter *type* [Macro]

Returns the object used to report conditions of the given *type*. This will be either a string, a function of two arguments (condition and stream) or `nil` if there is no report function. `setf` may be used with this form to change the report function for a condition type.

xcl:condition-handler *type* [Macro]

Returns the default handler for conditions of the given *type*. This will be a function of one argument or `nil` if there is no default handler. `setf` may be used with this form to change the default handler for a condition type.

conditions:make-condition *type* &**rest** *slot-initializations* [Function]

Calls the appropriate constructor function for the given type, passing along the given slot initializations to the constructor, and returning an instantiated condition.

The *slot-initializations* are given in alternating keyword/value pairs. eg,

```
(conditions:make-condition 'bad-food-color
   :food my-food
   :color my-color)
```

This function is provided mainly for writing subroutines that manufacture a condition to be signaled. Since all of the condition signalling functions can take a *type* and *slot-initializations*, it is usually easier to call them directly.

Signalling Conditions

xcl:*current-condition* [Variable]

This variable is bound by condition-signalling forms (`conditions:signal`, `error`, `cerror`, and `warn`) to the condition being signaled. This is especially useful in restart filters. The top-level value of `xcl:*current-condition*` is `nil`.

conditions:signal *datum* &**rest** *arguments* [Function]

Invokes the signal facility on a condition. If the condition is not handled, `conditions:signal` returns the condition object that was signaled.

If *datum* is a condition then that condition is used directly. In this case, it is an error for *arguments* to be non-`nil`.

If *datum* is a condition type, then the condition used is the result of doing

```
(apply #'conditions:make-condition
       datum arguments)
```

If *datum* is a string, then the condition used is the result of doing

```
(conditions:make-condition
   'conditions:simple-condition
   :format-string datum
   :format-arguments arguments).
```

~~If the condition is of type `xcl:serious-condition`, then `xcl:signal` will behave exactly like `error`, i.e., it will call~~

~~`xcl:debug` if the condition isn't handled, and will never return to its caller.~~

If `(typep` *condition* `conditions:*break-on-signals*)` is true, then the debugger will be entered prior to the signalling process. This is true for all other functions and macros that signal conditions, such as `warn`, `error`, `cerror`, `assert` and `check-type`.

conditions:*break-on-signals* [Variable]

This flag is primarily for use when debugging programs that do signaling. Its value is a type specifier.

When `(typep` *condition* `conditions:*break-on-signals*)` is true, then calls to `conditions:signal` and other functions that implicitly call `conditions:signal` will enter the debugger prior to signalling the condition. The `conditions:continue` restart may be used to continue with the normal signalling process.

The default value of this variable is `nil`.

Note: the variable `*break-on-warnings*` continues to be supported for compatibility, but `conditions:*break-on-signals*` offers that power and more. New code should not use `*break-on-warnings*`.

error *datum* &**rest** *arguments* [Function]

Like `conditions:signal` except if the condition is not handled, the debugger is called with the given condition, and `error` never returns.

datum is treated as in `conditions:signal`. If *datum* is a string, a conditon of type `conditions:simple-error` is made. This form is compatible with that described in Steele's *Common Lisp, the Language*.

cerror *proceed-format-string datum* &**rest** *arguments* [Function]

Like `error`, if the condition is not handled the debugger is called with the given condition. However, `cerror` enables the restart `conditions:continue`, which will simply return the condition being signalled from `cerror`.

datum is treated as in `error`. If *datum* is a condition, then that condition is used directly. In this case, *arguments* will be used only with the *proceed-format-string* and will not be used to initialize *datum*.

The *proceed-format-string* must be a string. Note that if *datum* is not a string, then the format arguments used by the *proceed-format-string* will still be the *arguments* (in the keyword format as specified). In this case, some care may be necessary to set up the *proceed-format-string* correctly. The `format` directive `~*` may be particularly useful in this situation.

The value returned by `cerror` is the condition which was signaled.

See Steele's *Common Lisp, the Language,* page 430 for examples of the use of `cerror`.

warn *datum* &**rest** *arguments* [Function]

Invokes the signal facility on a condition. If the condition is not handled, then the text of the warning is printed on `*error-output*`. If the variable `*break-on-warnings*` is true, then in addition to printing the warning, the debugger is entered using the function `break`. The value returned by `warn` is the condition that was signalled.

If *datum* is a condition, then that condition is used directly. In this case, if the condition is not of type `conditions:warning` or *arguments* is non-null, then an error of type `conditions:type-error` is signalled.

If *datum* is a condition type, then the condition used is the result of doing `(apply #'conditions:make-conditions` *datum arguments*`)`. This result must be of type `conditions:warning` or an error of type `conditions:type-error` is signalled.

If *datum* is a string, then the condition used is the result of `(conditions:make-conditions 'conditions:simple-warning :format-string` *datum* `:format-arguments` *arguments*`)`.

The precise mechanism for warning is as follows:

1) If `*break-on-warnings*` is true, the debugger will be entered. This feature is primarily for compatibility with old code: use of `conditions:*break-on-signals*` is preferred. If the break is continued using the `conditions:continue` restart, `warn` proceeds with step 2.

2) The warning condition is signalled. While it is being signalled, the `conditions:muffle-warning` restart is established for use by a handler to bypass further action by `warn`, i.e., to cause `warn` to immediately return.

3) The warning condition is reported to `*error-output*` by the `warn` function. Note that `warn` will indicate that the condition being signalled is a warning when it reports it, so there is no need for the condition to do so in its report method.

break-on-warnings [Variable]

check-type [Macro]

ecase [Macro]

ccase [Macro]

etypecase [Macro]

ctypecase [Macro]

assert [Macro]

All of the above behave as described in *Common Lisp: the Language*. The default clauses of `ecase` and `ccase` forms signal `conditions:simple-error` conditions. The default clauses of

`etypecase` and `ctypecase` forms signal `conditions:type-error` conditions. `assert` signals the `xcl:assertion-failed` condition. `ccase` and `ctypecase` set up a `conditions:store-value` restart.

Handling Conditions

conditions:handler-bind *bindings* &**rest** *forms* [Macro]

Executes the forms in a dynamic context where the given local handler *bindings* are in effect. The elements of *bindings* must take the form (*type-spec handler*). The handlers are bound in the order they are given, i.e., when searching for a handler, the error system will consider the leftmost binding in a particular `conditions:handler-bind` form first. However, while one of these handlers is running, none of the bindings established by the `conditions:handler-bind` will be in effect.

type must be a type specifier. To make a binding for several condition types, use (`or` *type1 type2* ...).

handler should evaluate to a function of one argument, a condition, to be used to handle a signalled condition during execution of the *forms*.

An example of the use of `conditions:handler-bind` appears at the end of the `conditions:restart-case` macro description.

conditions:handler-case *form* &**rest** *cases* [Macro]

xcl:condition-case *form* &**rest** *cases* [Macro]

Executes the given *form*. Each *case* has the form

(*type* ([*var*]) . *body*)

If a condition is signalled (and not handled by an intervening handler) during the execution of the form, and there is an appropriate clause—i.e., one for which

(`typep` *condition* '*type*)

is true—then control is transferred to the body of the relevant clause, binding *var*, if present, to the condition that was signaled. If no condition is signalled, then the values resulting from the *form* are returned by the `xcl:condition-case`. If the condition is not needed, `var` may be omitted.

Earlier clauses will be considered first by the error system. I.e.,

```
(xcl:condition-case form
   (cond1 ...)
   (cond2 ...))
```

is equivalent to

```
(xcl:condition-case
        (xcl:condition-case form
           (cond1 ...))
   (cond2 ...))
```

~~*type* may also be a list of types, in which case it will catch conditions of any of the specified types.~~

One may also specify an action to be taken if execution of *form* completes normally. This may be done by specifying a clause that has `:no-error` as its type. Such a clause, if provided, must be last. A `:no-error` clause looks like:

`(:no-error` *lambda-list* `.` *body*`)`

If execution of the form completes normally and there is a `:no-error` clause, the values produced by the form will be bound to variables in the clause's *lambda-list* and the *body* will be executed with none of the handler bindings in effect. In this case the value of the `xcl:condition-case` form is the value returned by the last form of the *body* of its `:no-error` clause. Having a :no-error clause is equivalent to wrapping `(mutiple-value-call #'(lambda` *lambda-list* `.` *body*`) ...)` around the `xcl:condition-case` form.

`conditions:handler-case` is synonymous with `xcl:condition-case`.

Examples:

```
(xcl:condition-case (/ x y)
   (division-by-zero () nil))

(xcl:condition-case (open *the-file*
                          :direction :input)
   (file-error (condition)
      (format t "~&Open failed: ~A~%" condition)))

(xcl:condition-case (some-user-function)
   (file-error (condition) condition)
   (division-by-zero () 0)
   ((or unbound-variable undefined-function) ()
      'unbound))

(xcl:condition-case (open my-file)
   (file-error ()
     (format *error-output* "Couldn't open ~S."
       my-file))
   (:no-error (stream)
     (open-more-files my-file stream) stream)))
```

Note the difference between `xcl:condition-case` and `conditions:handler-bind`. In `conditions:handler-bind`, you are specifying functions that will be called in the dynamic context of the condition signalling form. In `xcl:condition-case`, you are specifying continuations to be used instead of the original form if a condition of a particular type is signaled. These continuations will be executed in the same dynamic context as the original form.

conditions:ignore-errors &**body** *forms* [Macro]

Executes the forms in a context that handles conditions of type `error` by returning control to this form. If no error is signaled, all

values returned by the last form are returned by `conditions:ignore-errors`. Otherwise, the form returns the two values `nil` and the condition that was signaled. Synonym for

```
(xcl:condition-case (progn . forms)
   (error (condition)
      (values nil condition)).
```

~~`xcl.debug &optional` *datum* `&rest` *arguments*~~ [Function]

~~Enters the debugger with a given condition without signalling that condition. When the debugger is entered, it will announce the condition by invoking the condition's report function.~~

~~*datum* is treated the same as for `xcl:signal` except if *datum* is not specified, it defaults to `"Call to DEBUG"`.~~

~~This function will never directly return to its caller. Return can occur only by a special transfer of control, such as to a `catch`, `block`, `tagbody`, `xcl:proceed-case` or `xcl:catch-abort`.~~

conditions:invoke-debugger *condition* [Function]

Invokes the debugger with the given condition. This is intended to be used as a portable entry point to the debugger. For finer control over the debugging state, see the function `xcl:debugger`.

break &optional *format-string* `&`**rest** *format-arguments* [Function]

Enters the debugger with a simple condition with the given arguments. If no *format-string* is provided, it defaults to `"Break."` Computation may be continued by invoking the `conditions:continue` restart. If continued, `break` returns `nil`.

`break` is approximately:

```
(defun break (&optional (format-string "Break")
                &rest format-arguments)
   (conditions:restart-case (conditions:invoke-debugger
(conditions:make-conditions 'conditions:simple-condition
:format-string format-string :format-arguments format-
arguments)
      (conditions:continue ()
            :report "Return from BREAK."
         nil)))
```

Restarts

conditions:restart-case *expression* {(*case-name arglist* {*keyword value*}* {*form*}*)}* [Macro]

The *expression* is evaluated in a dynamic context where the case clauses have special meanings as points to which control may be transferred. If *expression* runs to completion, all values returned by the form are simply returned by the `conditions:restart-case` form. On the other hand, the computation of *expression* may choose to transfer control to one of the restart clauses. If a transfer to a clause occurs, the forms in the body of that clause will be

evaluated in the same dynamic context as the `conditions:restart-case` form, and any values returned by the last such form will be returned by the `conditions:restart-case` form.

A restart clause has the form given above:

(*case-name arglist* {*keyword value*}* {*form*}*)

The *case-name* may be `nil` or any symbol.

The *arglist* is a normal lambda list that will be bound and evaluated in the dynamic context of the `conditions:restart-case` form. They will use whatever values were provided by `conditions:invoke-restart` or `conditions:invoke-restart-interactively`. Definitions of these two functions appear later in this section.

The valid *keyword/value* pairs are:

`:filter` *expression*

expression should be suitable as an argument to the `function` special form. It defines a predicate of no arguments that determines if this clause is visible to `conditions:find-restart`. Default = true.

`:condition` *type*

Shorthand for a common special case of `:filter`. The following two *key/value* pairs are equivalent:

```
:condition foo

:filter
   (lambda ()
      (typep xcl:*current-condition*
             'foo))
```

`:interactive` *expression*

The *expression* must be a form suitable as an argument to `function`. (`function` *expression*) will be evaluated in the current lexical and dynamic environments. The result should be a function of no arguments which returns a list of values to be used by `conditions:invoke-restart-interactively`. This function will be called in the dynamic environment available prior to any restart attempt. Any interaction with the user should be done here and not in the body of the restart.

If there is no `:interactive` option specified and the restart is invoked interactively, no arguments will be supplied.

`:report` *expression*

The *expression* can either be a constant string or a form suitable as an argument to `function`.

If expression is not a string, (`function` *expression*) will be evaluated in the current lexical and dynamic environment. The

result should be a function of one argument, a stream, which will be called to report that restart. This function should print a short summary of the action that restart will take if invoked.

If *expression* is a string, it is a shorthand for `(lambda (s) (format s` *expression*`))`.

Only one of `:condition` or `:filter` may be specified. If no `:report` is specified, the *case-name* will be used. It is an error to have a null case name and no report function.

Examples:

```
(loop
  (conditions:restart-case
      (return (apply function some-args))
    (new-function (new-fn)
      :report "Use a different function."
      :interactive (lambda ()
                     (list (prompt-for 'function "Function:
")))
      (setq function new-fn))))

(loop
  (conditions:restart-case
      (return (apply function some-args))
    (nil (new-fn)
      :report "Use a different function."
      :interactive (lambda ()
                     (list (prompt-for 'function "Function:
")))
      (setq function new-fn))))

(conditions:restart-case (a-command-loop)
   (return-from-command-level ()
         :report
            (lambda (stream)
              (format stream "Return from command level ~D."
level))
      nil))

(loop
   (conditions:restart-case (another-computation)
      (conditions:continue () nil)))
```

The first and second examples are equivalent from the point of view of someone using the interactive debugger, but differ in one important aspect for non-interactive handling. If a handler "knows about" restart names, as in:

```
(when (conditions:find-restart 'new-function)
   (conditions:invoke-restart 'new-function the-
replacement))
```

then only the first example, and not the second, will have control transferred to its correction clause.

Here's a more complete example:

```
(let ((my-food 'milk)
      (my-color 'greenish-blue))
   (do ()
```

```
        ((not (food-colorable-p my-food
                                my-color)))
      (conditions:restart-case (error 'bad-food-color
                                :food my-food
                                :color my-color)
         (use-food (new-food)
            :report "Use another food."
            (setf my-food new-food))
         (use-color (new-color)
            :report "Use another color."
            (setf my-color new-color))))
   ;; We won't get to here until my-food
   ;; and my-color are compatible.
   (list my-food my-color))
```

Assuming that `use-food` and `use-color` have been defined as

```
(defun use-food (new-food)
  (invoke-restart 'use-food new-food))

(defun use-color (new-color)
  (invoke-restart 'use-color new-color))
```

then a handler can proceed from the error in either of two ways. It may correct the color or correct the food. For example:

```
#'(lambda (condition) ...
     ;; Corrects color
     (use-color 'white)  ...)
```

or

```
#'(lambda (condition) ...
     ;; Corrects food
     (use-food 'cheese) ...)
```

Here is an example using `conditions:handler-bind` and `conditions:restart-case`.

```
(conditions:handler-bind ((foo-error
                           #'(lambda (condition)
                               (conditions:use-value 7))))
   (conditions:restart-case (error 'foo-error)
      (conditions:use-value (x) (* x x))))
```

The above form returns `49`.

~~`xcl:define-proceed-function` *name* [Macro]~~
~~{*keyword value*}*~~
~~{*variable*}*~~

~~Valid *keyword/value* pairs are the same as those which are defined for the `xcl:proceed-case` special form. That is, `:filter`, `:filter-function`, `:condition`, `:report`, and `:report-function`. The filter and report functions specified in a `xcl:define-proceed-function` form will be used for `xcl:proceed-case` clauses with the same name that do not specify their own filter or report functions, respectively.~~

~~This form defines a function called `name` which will invoke a proceed case with the same name. The proceed function takes optional arguments which are given by the *variables* specification. The parameter list for the proceed function will look like~~

~~(&optional . *variables*)~~

~~The only thing that a proceed function really does is collect values to be passed on to a proceed case clause.~~

~~Each element of *variables* has the form *variable-name* or (*variable-name initial-value*). If *initial-value* is not supplied, it defaults to `nil`.~~

~~For example, here are some possible proceed functions which might be useful in conjunction with the `bad-food-color` error we used as an example earlier:~~

```
(xcl:define-proceed-function use-food
       :report "Use another food."
    (food (read-typed-object 'food
                             "Food to use instead: ")))

(xcl:define-proceed-function use-color
       :report "Change the food's color."
    (color
       (read-typed-object 'food
                "Color to make the food: ")))

(defun maybe-use-water (condition)
    ;; A sample handler
    (when (eq (bad-food-color-food condition)
              'milk)
       (use-food 'water)))

(xcl:handler-bind ((bad-food-color
                                  #'maybe-use-water))
    ...)
```

~~If a named proceed function is invoked in a context in which there is no active proceed case by that name, the proceed function simply returns `nil`. So, for example, in each of the following pairs of handlers, the first is equivalent to the second but less efficient:~~

```
#'(lambda (condition)              ; OK, but slow
      (when (xcl:find-proceed-case 'use-food)
         (use-food 'milk)))
#'(lambda (condition)              ; Preferred
      (use-food 'milk))

#'(lambda (condition)
      (cond ((xcl:find-proceed-case 'use-food)
             (use-food 'chocolate))
            ((xcl:find-proceed-case 'use-color)
             (use-color 'orange))))
#'(lambda (condition)
      (use-food 'chocolate)
      (use-color 'orange))
```

conditions:restart-bind ({(*name function* {*keyword value*}*)}* {*form*}* [Macro]

Executes the *form*s in a dynamic context where the given restart bindings are in effect.

name may be `nil` to indicate an anonymous restart, or some other symbol to indicate a named restart.

function will be evaluated in the current lexical and dynamic contexts and should produce a function of no arguments to be used to perform the restart. This function will be called when that restart is activated by `conditions:invoke-restart` or `conditions:invoke-restart-interactively`. Note that unlike `conditions:restart-case`, invoking the restart does not automatically transfer control back to the contour in which it was established. If that is appropriate for that restart it is up to the individual restart function to do this.

The valid *keyword*/*value* pairs are:

`:interactive-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of no arguments that will construct the list of values to be used by `conditions:invoke-restart-interactively`.

`:report-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of one argument, a stream, that will be used to report that restart.

`:filter-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of no arguments that will be used to determine if the given restart is currently active.

This form is a more primitive way of establishing restarts than *conditions:restart-case*. It is expected that *conditions:restart-case* will be sufficient for most uses of the restart facility. An example of where the more general facility provided by *conditions:restart-bind* may be useful is:

```
(conditions:restart-bind ((nil #'(lambda () (expunge-directory the-dir)) :report-function #'(lambda (stream) (format stream "Expunge ~A." (directory-namestring the-dir))))) (cerror "Try this file operation again." 'directory-full :directory the-dir))
```

In this case, a restart is provided that allows the user to expunge the full directory and return to the debugger after doing so. He can then try some other restart, such as `conditions:continue` to retry the failed operation.

conditions:compute-restarts [Function]

Uses the dynamic state of the program to compute a list of *restarts*.

Each restart object represents a point in the current dynamic state of the program to which control may be transferred. The only operations that Lisp defines for such objects are:

`conditions:restart-name`,
`conditions:find-restart`,
`conditions:invoke-restart`, `conditions:invoke-restart-interactively`,
`princ`, and
`prin1`,

to identify an object as a restart using `(typep` *x* `'conditions:restart)`, and standard Lisp operations that work for all objects, such as `eq`, `eql`, `describe`, etc.

The list which results from a call to `conditions:compute-restarts` is ordered so that the innermost (ie, more-recently established) restarts are nearer the head of the list.

Note also that `conditions:compute-restarts` returns *all* valid restarts, even if some of them have the same name as others and therefore would not be found by `conditions:find-restart`.

It is an error to modify the list returned by `conditions:compute-restarts`.

conditions:restart-name *restart* [Function]

Returns the name of the given *restart*, or `nil` if it is not named.

~~`xcl:default-proceed-test` *proceed-case-name* [Macro]~~

~~Returns the default filter function for proceed-cases with the given *proceed-case-name*. May be used with `setf` to change it.~~

~~`xcl:default-proceed-report` *proceed-case-name* [Macro]~~

~~Returns the default report function for proceed-cases with the given *proceed-case-name*. This may be a string or a function just as for condition types. May be used with `setf` to change it.~~

conditions:find-restart *identifier* [Function]

Searches for a restart by the given *identifier* which is in the current dynamic environment.

If *identifier* is a symbol, then the innermost (ie, most recently established) restart with that name that is active is returned. `nil` is returned if no such restart is found.

If *identifier* is a restart object, then it is simply returned unless it is not currently valid for use. In that case, `nil` is returned.

When searching for a matching restart, the filter function, if any, of potential matches will be called to see if they are active. If it returns

`nil`, then the restart is considered to not have been seen and the search for a match continues.

Although anonymous restarts have a name of `nil`, it is an error for the symbol `nil` to be given as an *identifier* to this function. If it is approriate to search for anonymous restarts, you should use `conditions:compute-restarts` instead.

conditions:invoke-restart *restart* &**rest** *values* [*Function*]

Calls the function associated with the given *restart*, passing the *values* as arguments. The *restart* must be a restart object or the non-null name of a restart which is valid in the current dynamic context. If an argument is not valid, an error of type `conditions:control-error` will be signalled.

~~If the argument is a named proceed case that has a corresponding proceed function, `xcl:invoke-proceed-case` will do the optional argument resolution specified by that function before transferring control to the proceed case.~~

conditions:invoke-restart-interactively *restart* [Function]

Calls the function associated with the given *restart*, providing for any necessary arguments. The *restart* must be a restart object or the non-null name or a restart which is valid in the current dynamic context. If the *restart* is not valid, an error of type `conditions:control-error` will be signalled.

`conditions:invoke-restart-interactively` will first call the *restart*'s interactive function as specified by the `:interactive` keyword of `conditions:restart-case` or the `:interactive-function` keyword of `conditions:restart-bind`. The interactive function should return a list of values to be passed as arguments to the *restart*. This list must be at least as long as the number of required arguments that the *restart* has.

If the *restart* has no interactive function, no arguments will be passed to the restart function. It is an error for a restart to require arguments but not have an interactive function.

Once the arguments have been determined, `conditions:invoke-restart-interactively` will simply do `(apply #'conditions:invoke-restart` *restart arguments*`)`.

conditions:with-simple-restart `(`*name format-string* `{`*format-arguments*`}*)` `{`*form*`}*`

[Macro]

This is a shorthand for one of the most common uses of `conditions:restart-case`.

If the *restart* designated by *name* is not invoked while executing the *form*s, all values produced by the last *form* are returned. If the restart established by `conditions:with-simple-restart` is

invoked, control is transferred to the `conditions:with-simple-restart` form, which immediately returns the two values `nil` and `t`.

It is permissible for *name* to be `nil`. In that case, an anonymous restart is established.

`conditions:with-simple-restart` is essentially:

```
(defmacro conditions:with-simple-restart
    ((restart-name format-string
      &rest format-arguments)
     &body forms)
  `(conditions:restart-case (progn ,@forms)
     (,restart-name ()
         :report (lambda (stream)
                   (format stream
                     ,format-string
                     ,@format-arguments))
       (values nil t))))
```

Example:

```
(defun read-eval-print-loop (level)
  (conditions:with-simple-restart
      (conditions:abort "Exit command level ~D." level)
    (loop
      (conditions:with-simple-restart
          (conditions:abort "Return to command level ~D."
level)
        (print (eval (read)))))))
```

xcl:catch-abort *print-form* &**body** *forms* [Macro]

Like `conditions:with-simple-restart`, but always uses the name `conditions:abort` .

`xcl:catch-abort` could be defined by:

```
(defmacro xcl:catch-abort (print-form
                           &body forms)
   `(conditions:with-simple-restart
        (conditions:abort ,print-form)
      ,@forms))
```

conditions:abort [Function]

This function transfers control to the nearest active restart named `conditions:abort`. If there is none, this function signals an error of type `conditions:control-error`.

~~`xcl:abort` could be defined by:~~

```
~~(define-proceed-function xcl:abort
    :report "Abort")~~
```

conditions:continue [Function]

This function transfers control to the nearest active restart named `conditions:continue`. If none exists it simply returns `nil`.

The `conditions:continue` restart is generally part of simple protocols where there is a single "obvious" way to continue, such as in `break` and `cerror`.

NB: `conditions:continue` replaces `xcl:proceed`.

~~`xcl:proceed &optional` *condition* [Function]~~

~~This is a predefined proceed function. It is used by such functions as `break`, `cerror`, etc.~~

conditions:muffle-warning [Function]

This function transfers control to the nearest active restart named `conditions:muffle-warning`. If none exists, an error of type `conditions:control-error` is signalled.

`warn` sets up this restart so that handlers of `conditions:warning` conditions have a way to tell `warn` that the warning has been dealt with and that no further action is warranted.

conditions:use-value *new-value* [Function]

This function transfers control (and one value) to the nearest active restart named `conditions:use-value`. If no such restart exists, this function simply returns `nil`.

The `conditions:use-value` restart is generally used by handlers trying to recover from errors of types such as `conditions:cell-error`, where the handler may wish to supply a replacement datum for one-time use.

conditions:store-value *new-value* [Function]

This function transfers control (and one value) to the nearest active restart named `conditions:store-value`. If no such restart exists, this function simply returns `nil`.

The `conditions:use-value` restart is generally used by handlers trying to recover from errors of types such as `conditions:cell-error`, where the handler may wish to supply a replacement datum to be stored in the offending cell.

[This page intentionally left blank]

COVER TEXT:

NOTE: USE STANDARD 8-1/2X11 BINDER COVERS, 3-ring, D-ring binder, 2 1/4 inch spine

ENVOS NAME/LOGO HERE
(probably sticker)

ARTIFICIAL INTELLIGENCE SYSTEMS

HELVETICA
BOLD ITALIC

LISP RELEASE NOTES

MEDLEY RELEASE

MODERN
BOLD ITALIC

ENVOS PART NO. XXXXX
(probably sticker)

SPINE (2-1/4 INCH):

ENVOS

ARTIFICIAL INTELLIGENCE SYSTEMS

LISP RELEASE NOTES

MEDLEY RELEASE

HELVETICA
BOLD ITALIC

MODERN
BOLD ITALIC

LISP RELEASE NOTES

Medley Release
September 1988

LISP RELEASE NOTES

Medley Release

September 1988

This manual is set in Modern and Terminal typefaces with text written and formatted on Xerox Artificial Intelligence workstations. Xerox laser printers were used to produce text masters.

INDEX

LIST OF TABLES

[This page intentionally left blank]

PREFACE

The *Lisp Release Notes* provide current information about the Lisp software development environment. You will find the following information in these *Notes*:

- An overview of significant extensions to the Common Lisp language.
- Descriptions of new features that enhance the integration and implementation of Common Lisp into the Lisp environment.
- A summary of changes made in the Library modules, in the Sketch and TEdit tools, and in the 1108 and 1186 User's Guides.
- Discussions of how specific Common Lisp features have affected the Interlisp-D language.
- Notes reflecting the changes made to Interlisp-D, independent of Common Lisp.
- Known restrictions.

How the Release Notes are Organized

The *Lisp Release Notes* are organized as follows:

Chapter 1, Introduction, summarizes the Medley release enhancements.

Chapter 2, Notes and Cautions, highlights significant Medley and Lyric changes in the Lisp environment.

Chapter 3, Common Lisp/Interlisp-D Integration, discusses how the integration of Common Lisp into the Lisp environment affects Interlisp features.

Chapter 4, Changes to Interlisp-D in Lyric/Medley, outlines changes that have taken place in Interlisp-D and its environment during the Lyric and Medley releases. These changes are primarily independent of Common Lisp integration.

> Chapters 3 and 4 are organized to parallel the *Interlisp-D Reference Manual* as closely as possible. To make it easy to use these chapters with the *IRM,* the following conventions are used:
>
> Information is organized by *Interlisp-D Reference Manual* volume and section. The *IRM* section level headings are maintained to aid in cross-referencing.

Chapter 5, Library Modules, is a synopsis of the changes to the Lisp Library Modules.

Chapter 6, User's Guides, is a collection of release notes on the 1108 and 1186 User's Guides; *A User's Guide to Sketch*, and *A User's Guide to TEdit*.

Chapter 7, Common Lisp Implementation, describes improved features that integrate Common Lisp into the environment.

Five Appendices contain documentation of newly integrated system features:

Appendix A, The Exec, describes Lisp's Exec.

Appendix B, SEdit, describes the Lisp structure editor.

Appendix C, ICONW, describes the Lisp feature for building display icons .

Appendix D, Free Menu, describes Lisp's flexible menu feature.

Appendix E, Error System, describes error conditions and recovery.

Notational Conventions

Conventions used in the *Lisp Release Notes* include the following:

Names of Interlisp functions, macros and variables are shown in **BOLD UPPERCASE**; their arguments are in *ITALICS*.

Names of Common Lisp functions, macros and variables are shown in **bold lowercase**; their arguments are in *italics.*

A backslash (\) character preceding a function or variable name signifies that it is a property of the system.

Examples are shown in `terminal 10`.

Text shown with ~~StrikeThru~~ is information that no longer applies.

Text shown with revision bars in the right margin is information that has been added or modified since the last release.

References to the *Interlisp-D Reference Manual*, or *IRM*, are used throughout this manual.

How to Use The Release Notes

The *Lisp Release Notes* contain current information on the Lisp environment. The Medley release enhances the Lyric release with new features and corrections to over 450 known Lyric bugs. Because Medley primarily contains additions to Lyric, these *Release Notes* have been written to include Lyric information that applies in Medley.

These *Lisp Release Notes* replace the *Lyric Release Notes*. The descriptions contained within these *Notes* are closely interwoven with functions, variables and other concepts discussed in the *Interlisp-D Reference Manual*, or *IRM*. Chapters 3 and 4 of these *Notes* closely parallel the *IRM,* preserving section headings with respect to order and numbering. You might find it helpful to go through the *Release Notes* and the *IRM* together, marking the *IRM* sections that have new information. Later when you consult the *IRM* you will know which sections require you to read the analogous section of the *Release Notes* .

Related Literature

We recommend that you use the *Lisp Release Notes* as a supplement to the following publications:

Interlisp-D Reference Manual, Volumes I through III, Koto Release, 1985.

Common Lisp, the Language, by Guy L. Steele Jr., Digital Press, 1984.

Common Lisp Implementation Notes, Lyric Release, 1987.

Lisp Documentation Tools, (includes *"A User's Guide to TEdit" and "A User's Guide to Sketch"),* Lyric Release, 1987.

Lisp Library Modules, Medley Release, 1988.

Medley 1.0-S User's Guide, Medley Release, 1988.

[This page intentionally left blank]

PRINTING SPECIFICATIONS

LISP RELEASE NOTES, MEDLEY RELEASE
AND
LISP LIBRARY MODULES, MEDLEY RELEASE

Special Instructions: >>3-hole punch (error tolerances: + or - 1/16 inch)<<

PRINTING

Printing Method: **>>**offset**<<**

Paper Weight: >>60 lb. **<<**

Paper Type (Finish): >> Matte<<

Paper Color: >>White<<

Paper Texture: >>Smooth<<

Paper Opacity: 92 (no show through)

Paper Size: >>8-1/2 X 11<<

Exceptions (e.g., oversize diagrams): >>none**<<**

Special Instructions: >><<

Number of Pages: >><<

PAPER

1. INTRODUCTION

The *Lisp Release Notes* contain information from both the Lyric and Medley releases, including descriptions of all Lyric bug fixes. Medley additions are indicated with revision bars in the right margin.

Summary of Medley Changes

The Medley release is currently provided on two platforms, Xerox 1100 series workstations using Medley 1.0, and Sun 3 workstations using Medley 1.0-S. Medley 1.0 and Medley 1.0-S are compatible with each other and will let you develop software on either platform. Source and compiled files are transferable between the two platforms. Sysouts developed on Xerox workstations can also be run on the Sun 3. Sysouts made on the Sun 3, however, cannot be run on Xerox workstations.

The Medley release enhances the Lyric release and fixes over 450 known Lyric bugs. Medley adds new features, improves Common Lisp implementation, and improves overall reliability of the Lisp sysout. Specific enhancements include:

- The COMPILER contains many added optimizations and numerous bug fixes.
- The DEBUGGER evaluates lexical variables. Lexical variables can now be contained in interpreted code.
- DFASL files now behave at the level of Interlisp-D compiled files. COMS are contained in DFASLs so that the system loads a DFASL file only once.
- The SEDIT code editor is more robust and better integrated with the environment.
- Common Lisp comments are preserved during loading. During MAKEFILE, comments can be written out with just semicolons.
- The new ERROR SYSTEM is compatible with the most recent standard defined for Common Lisp error systems.
- TEdit contains numerous bug fixes.
- MASTERSCOPE contains Common Lisp query support allowing you to ask questions about Common Lisp code that could previously be asked only of Interlisp-D code. Currently, questions specific to Common Lisp constructs are not supported.
- RS232 contains may bug fixes that improve the reliability of data transfer and the addition of various debugging tools.
- TCP/IP now contains many bug fixes including UNIX file interface and directory enumeration.

- A new System Tool lets you fetch sysouts from TCP hosts.
- NS Random Access is now supported.
- A new File Browser user interface now supports file sorting by dates. The new interface includes the ability to stop in the middle of operations.
- The Medley sysout is about the same size as the Lyric sysout.

In addition, Medley on the Sun 3 workstation offers the following new features:

- The UnixChat library module allows you to communicate with a UNIX shell on your own host.
- The UnixComm library module allows you to start a Unix process on a Sun workstation and provides an interface to the SunOS operating system.
- The ability to suspend Medley and use UNIX as a background process is provided.

[This page intentionally left blank]

2. NOTES AND CAUTIONS

This section contains notes and cautions that apply in Lyric and Medley. Medley notes are indicated with revision bars in the right margin. Text shown with ~~StrikeThru~~ is that information from the Lyric release that no longer applies.

Changes and Cautions in the Medley Release

- The Medley Release is currently provided on two platforms, the Xerox 1100 series workstations and selected Sun workstations. File structure for the 1108/09/86 remains the same. For Sun workstations, UNIX file structure is supported. See the *Medley 1.0-S User's Guide* for details.
- Files compiled in Medley cannot be loaded back into Lyric. Medley-compiled .LCOM and .DFASL files will produce an error message when loaded into Lyric. (Lyric-compiled .LCOM and .DFASL files can be loaded and run in Medley.) If you need to run a Medley file in Lyric, load the source file and use the Lyric compiler.
- SEdit and definers now support four-semicolon and balanced comments. Print support for these new types of comments is also provided. For details, see "TextModules" in the *Lisp Library Modules* manual, and "SEdit" in Appendix B of this manual.
- Medley and Lyric can both be installed on one machine.

Changes and Cautions in the Lyric Release

- Koto and Lyric cannot both be supported on one machine.
- You must have Services 10.0 installed on your printers to correctly print TEdit files.
- Interlisp DMACROs are not visible to Common Lisp. If a symbol has both a function definition and a DMACRO property, the compiler assumes that the DMACRO is an optimizer for the old Interlisp compiler and ignores it.
- The Common Lisp functions found in *Common Lisp, The Language,* section 25.4.2, "Other environmental Inquiries" (e.g., LISP-IMPLEMENTATION-TYPE) are in the COMMON LISP (CL:) package.
- Both Medley and Lyric use the new type of Executive, and both sysouts contain the ability to spawn multiple executive processes. The default executive is Common Lisp, not

Interlisp. The old Executive (the "Programmer's Assistant") is not available in Medley.

You should be particularly careful in the new Executives when typing file names, as some file name delimiters now have syntactic significance in the new readtables. In particular, the character colon (:) used in NS file server names is a package delimiter in all new Executives, and the version delimiter semi-colon (;) is a comment character in the Common Lisp Executives. If you type a file name in the form of a symbol to an Exec, you must escape the special characters, or use the multiple escape character around the whole name. For example, in a Common Lisp Exec you might type

{FS\:Me\:Company}<Fred>Stuff.tedit\;3

or

|{FS:Me:Company}<Fred>Stuff.tedit;3|,

which are equivalent, except that the former is read as all upper case (Common Lisp Exec's read case-insensitively). This caution should also be noted when copy-selecting file names out of a File Browser.

It is recommended that you type file names as strings whenever possible, as virtually all system interfaces accept strings instead of symbols. Two notable exceptions are MAKEFILE and TEDIT, which require symbols when naming files.

These escaping rules apply *only* to file names typed to an Executive (or in general, a Lisp reader). Individual tools that prompt for a file name read the name as a string, so escape characters need not (and should not) be typed. In particular, this is true for the prompt windows of TEdit and File Browser, and the prompt for an Init file when a system with no local Init file is started up.

- A new error system, based on the current Common Lisp proposed error standard, replaces the old Interlisp error system.
- The **!EVAL** debugger command no longer exists and the = and -> break commands are no longer supported..
- The function **ERRORN** no longer exists and **ERRORTYPELIST** is no longer supported. See Chapter 3, Common Lisp /Interlisp Integration, section 14.10 "Error List" for Interlisp errors that are no longer supported.
- The Lyric release contained a new compiler and compiled code format, .DFASL (FASt Loading) files. The old compiler is still available and produces files in the old format, but with extension .LCOM. The old compiler will not be available in future releases.
- Files produced by the Lyric File Manager cannot be loaded into previous releases of the system. Files compiled in Koto cannot be loaded into Lyric.

- SETQ from the exec does not interact with the File Manager, nor does it print (var reset) (except in the "Programmer's Assistant").
- DWIM/CLISP: CLISP infix is no longer fully supported; users should dwimify old Koto code before running it in Lyric. Additionally, WITH constructs using "←" and BIND constructs in the form of an atom A←B need to be dwimified.
- The functions BREAKDOWN and BRKDWNRESULTS as well as the variables, BRKDWNTYPE and BRKDWNTYPES have been removed from the environment. The Lisp Library Module, SPY supersedes BREAKDOWN.
- The file system supports having multiple streams opened on a single file at one time. This means that the input/output functions accept only streams as arguments, not symbols naming files. This has several implications for Interlisp programmers, one being that the function **CLOSEALL** is no longer implemented. See the Chapter 3, Common Lisp/Interlisp Integration, Streams and Files section, for details.
- Windows cannot be used interchangeably with streams in Common Lisp functions. If you need to use a window in the middle of a Common Lisp function, use (IL:GETSTREAM window) to get the associated display stream.
- Loading CPM-format floppies is very slow in Lyric. CPM-format floppies are not supported in Medley.
- The default Interlisp readtable has been modified for compatibility with Common Lisp. The characters colon (:), hash (#) and vertical bar (|) have different meaning. The File Manager gives a choice of reader environments in which to write files, and remembers which one was used for each file.
- READ/PRINT consistency: Old Interlisp code that used **READ** and **PRINT** without being careful about using a particular readtable may need to be fixed.
- The Interlisp function **SKREAD** defaults its readtable argument to the current readtable, viz., the value of ***READTABLE***, rather than **FILERDTBL**.
- FREEMENU and ICONW, formerly Library modules, are included in the Lisp.sysout in Lyric and Medley.
- The Lyric Lisp editor, SEdit, has been modified in Medley. DEdit is now a library module.
- Revised fonts: Lyric revised the naming convention for font files, and printer width files had corrected line leading information. ~~Old Koto fonts can still be used, but you are encouraged to start using the new fonts as soon as practicable.~~ Medley and Lyric fonts are completely compatible.
- Lyric image objects are now stored on files in a way that cannot always be read into Koto. (Lyric, on the other hand, can read Koto image objects.) This means, for example, that you may not be able to share TEdit files or sketches containing image objects between Koto and Lyric.

- The field names for the CURSOR datatype have been changed.
- Masterscope has been removed from the standard environment. If you wish to use it, load the Masterscope Library module.
- Pattern matching is no longer a part of the standard environment. Pattern matching can be found in the Lisp Library Module, Match.
- PRESS fonts are not part of the standard Lisp environment. PRESS is now available as a Library Module.
- In Lyric, the Library module TCP/IP does not work on 1186 workstations that have *both* IOPs with part number 140K03030 *and* "old" ROMs. The problem is not with the IOP board per se, rather it's a problem with the IOP's ROMs. If TCP/IP doesn't work on your 1186 you should check your IOP board revision. If you have the old IOP you may need to replace the ROMs before you can use TCP/IP, contact your service representative.

  TCP/IP does work with newer IOPs—part number 140K05560.

  If you attempt to Teleraid a Lyric sysout from a Koto sysout, you should be aware of the following:

  1. All symbols will be read as if they were in the INTERLISP package and you can only type a subset of the IL symbols to it.

  2. Teleraid will not understand certain Common Lisp datatypes, such as CHARACTER and strings.

[This page intentionally blank]

3. COMMON LISP/INTERLISP-D INTEGRATION

NOTE: Chapter 3 is organized to correspond to the original *Interlisp-D Reference Manual*, and explains changes related to how Common Lisp affects Interlisp-D in your Lisp software development environment. To make it easy to use this chapter with the *IRM,* information is organized by *IRM* volume and section numbers. Section headings from the *IRM* are maintained to aid in cross-referencing.

Lyric information as well as Medley release enhancements are included. Medley additions are indicated with revision bars in the right margin.

VOLUME I—LANGUAGE

Chapter 2 Litatoms

(2.1)

What Interlisp calls a "LITATOM" is the same as what Common Lisp calls a "SYMBOL." Symbols are partitioned into separate name spaces called packages. When you type a string of characters, the resulting symbol is searched for in the "current package." A colon in the symbol separates a package name from a symbol name; for example, the string of characters "CL:AREF" denotes the symbol AREF accessible in the package CL. For a full discussion, see Guy Steele's *Common Lisp, the Language*.

All the functions in this section that create symbols do so in the INTERLISP package (IL), which is also where all the symbols in the *Interlisp-D Reference Manual* are found. Note that this is true even in cases where you might not expect it. For example, U-CASE returns a symbol in the INTERLISP package, even when its argument is in some other package; similarly with L-CASE and SUBATOM. In most cases, this is the right thing for an Interlisp program; e.g., U-CASE in some sense returns a "canonical" symbol that one might pass to a SELECTQ, regardless of which executive it was typed in. However, to perform symbol manipulations that preserve package information, you should use the appropriate Common Lisp functions (See *Common Lisp the Language*, Chapter 11, Packages and Chapter 18, Strings).

Symbols read under an old Interlisp readtable are also searched for in the INTERLISP package. See Section 25.8, Readtables, for more details.

Section 2.1 Using Litatoms as Variables

(I:2.3)

(**BOUNDP** *VAR*) [Function]

The Interlisp interpreter has been modified to consider any symbol bound to the distinguished symbol **NOBIND** to be unbound. It will signal an UNBOUND-VARIABLE condition on encountering references to such symbols. In prior releases, the interpreter only considered a symbol unbound if it had no dynamic binding and in addition its top-level value was **NOBIND**.

For most user code, this change has no effect, as it is unusual to bind a variable to the particular value **NOBIND** and still deliberately want the variable to be considered bound. However, it is a particular problem when an interpreted Interlisp function is passed to the function **MAPATOMS**. Since **NOBIND** is a symbol, it will eventually be passed as an argument to the interpreted function. The first reference to that argument within the function will signal an error.

A work-around for this problem is to use a Common Lisp function instead. Calls to this function will invoke the Common Lisp interpreter which will treat the argument as a local, not special, variable. Thus, no error will be signaled. Alternatively, one could include the argument to the Interlisp function in a **LOCALVARS** declaration and then compile the function before passing it to **MAPATOMS**. This has the advantage of significantly speeding up the **MAPATOMS** call.

Section 2.3 Property Lists

(I:2.6)

The value returned from the function **REMPROP** has been changed in one case:

(**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**.

Section 2.4 Print Names

(I:2.7)

The print functions now qualify the name of a symbol with a package prefix if the symbol is not accessible in the current package. The Interlisp "PRIN1" print name of a symbol does not include the package name.

(I:2.10)

The **GENSYM** function in Interlisp creates symbols interned in the INTERLISP package. The Common Lisp **CL:GENSYM** function creates uninterned symbols.

(I:2.11)

(**MAPATOMS** *FN*) [Function]

See the note for **BOUNDP** above.

Section 2.5 Characters

A "character" in Interlisp is different from the type "character" in Common Lisp. In Common Lisp, "character" is a distinguished data type satisfying the predicate **CL:CHARACTERP**. In Interlisp, a "character" is a single-character symbol, not distinguishable from the type symbol (litatom). Interlisp also uses a more efficient object termed "character code", which is indistinguishable from the type integer.

Interlisp functions that take as an argument a "character" or "character code" do not in general accept Common Lisp characters. Similarly, an Interlisp "character" or "character code" is not acceptable to a Common Lisp function that operates on characters. However, since Common Lisp characters are a distinguished datatype, Interlisp string-manipulation functions are willing to accept them any place that a "string or symbol" is acceptable; the character object is treated as a single-character string.

To convert an Interlisp character code *n* to a Common Lisp character, evaluate (**CL:CODE-CHAR** *n*). To convert a Common Lisp character to an Interlisp character code, evaluate (**CL:CHAR-CODE** *n*). For character literals, where in Interlisp one would write (**CHARCODE** *x*), to get the equivalent Common Lisp character one writes #*x*. In this syntax, *x* can be any character or string acceptable to **CHARCODE**; e.g., #\GREEK-A.

Chapter 4 Strings

(I:4.1)

Interlisp strings are a subtype of Common Lisp strings. The functions in this chapter accept Common Lisp strings, and produce strings that can be passed to Common Lisp string manipulation functions.

Chapter 5 Arrays

Interlisp arrays and Common Lisp arrays are disjoint data types. Interlisp arrays are not acceptable arguments to Common Lisp array functions, and vice versa. There are no functions that convert between the two kinds of arrays.

Chapter 6 Hash Arrays

Interlisp hash arrays and Common Lisp hash tables are the same data type, so Interlisp and Common Lisp hash array functions may be freely intermixed. However, some of the arguments are different; e.g., the order of arguments to the map functions in **IL:MAPHASH** and **CL:MAPHASH** differ. The extra functionality of specifying your own hashing function is available only from Interlisp **HASHARRAY**, not **CL:MAKE-HASH-TABLE** , though the latter does supply the three built-in types specified by *Common Lisp, the Language*.

Chapter 7 Numbers and Arithmetic Functions

(I:7.2)

The addition of Common Lisp data structures within the Lisp environment means that there are some invariants which used to be true for anything in the environment that are no longer true.

For example, in Interlisp, there were two kinds of numbers: integer and floating. With Common Lisp, there are additional kinds of numbers, namely ratios and complex numbers, both of which satisfy the Interlisp predicate **NUMBERP**. Thus, **NUMBERP** is no longer the simple union of **FIXP** and **FLOATP**. It used to be that a program containing

```
(if (NUMBERP X)
  then (if (FIXP X)
           then ...assume X is an integer ...
         else ...can assume X is floating point...))
```

would be correct in Interlisp. However, this is no longer true; this program will not deal correctly with ratios or complex numbers, which are **NUMBERP** but neither **FIXP** nor **FLOATP**.

Section 7.2 Integer Arithmetic

When typing to a *new* Interlisp Executive, the input syntax for integers of radix other than 8 or 10 has been changed to match that of Common Lisp. Use # instead of |, e.g., #b10101 is the new syntax for binary numbers, #x1A90 for hexadecimal, etc. Suffix Q is still recognized as specifying octal radix, but you can also use Common Lisp's #o syntax.

(I:7.4)

In the Lyric release, the FASL machinery would handle some positive literals incorrectly, reading them back as negative numbers. The numbers handled incorrectly were those numbers x greater than 2**31-1 for which (mod (integer-length x) 8) was zero. The Medley release fixes this situation. Any files containing such numbers should be recompiled.

Chapter 10 Function Definition, Manipulation, and Evaluation

Section 10.1 Function Types

All Interlisp **NLAMBDA**s appear to be macros from Common Lisp's point of view. This is discussed at greater length in *Common Lisp Impementation Notes*, Chapter 8, Macros**.**

Section 10.6 Macros

(**EXPANDMACRO** *EXP QUIETFLG — —*) [Function]

EXPANDMACRO only works on Interlisp macros, those appearing on the MACRO, BYTEMACRO or DMACRO properties of symbols. Use **CL:MACROEXPAND-1** to expand Common Lisp macros and those Interlisp macros that are visible to the Common Lisp compiler and interpreter.

Section 10.6.1 DEFMACRO

(I:10.24)

Common Lisp does not permit a symbol to simultaneously name a function and a macro. In Lyric, this restriction also applies to Interlisp macros defined by **DEFMACRO**. That is, evaluating **DEFMACRO** for a symbol automatically removes any function definition for the symbol. Thus, if your purpose for using a macro is to make a function compile in a special way, you should instead use the new form **XCL:DEFOPTIMIZER**, which affects only compilation. The *Xerox Common Lisp Implementation Notes* describe **XCL:DEFOPTIMIZER**.

Interlisp DMACRO properties have typically been used for implementation-specific optimizations. They are not subject to the above restriction on function definition. However, if a symbol has both a function definition and a DMACRO property, the Lisp compiler assumes that the DMACRO was intended as an optimizer for the old Interlisp compiler and ignores it.

Chapter 11 Stack Functions

Section 11.1 The Spaghetti Stack

Stack pointers now print in the form

`#<Stackp` address/framename`>`.

Some restrictions were placed on spaghetti stack manipulations in order to integrate reasonably with Common Lisp's **CL:CATCH** and **CL:THROW**. In Lyric, it is an error to return to the same frame twice, or to return to a frame that has been unwound through. This means, for example, that if you save a stack pointer to one of your ancestor frames, then perform a **CL:THROW** or **RETFROM** that returns "around" that frame, i.e., to an ancestor of that frame, then

the stack pointer is no longer valid, and any attempt to use it signals an error "Stack Pointer has been released". It is also an error to attempt to return to a frame in a different process, using **RETFROM**, **RETTO**, etc.

The existence of spaghetti stacks raises the issue of under what circumstances the cleanup forms of **CL:UNWIND-PROTECT** are performed. In Lisp, **CL:THROW** always runs the cleanup forms of any **CL:UNWIND-PROTECT** it passes. Thanks to the integration of **CL:UNWIND-PROTECT** with **RESETLST** and the other Interlisp context-saving functions, **CL:THROW** also runs the cleanup forms of any **RESETLST** it passes. The Interlisp control transfer constructs **RETFROM**, **RETTO**, **RETEVAL** and **RETAPPLY** also run the cleanup forms in the analogous case, viz., when returning to a direct ancestor of the current frame. This is a significant improvement over prior releases, where **RETFROM** never ran any cleanup forms at all.

In the case of **RETFROM**, etc, returning to a non-ancestor, the cleanup forms are run for any frames that are being abandoned as a result of transferring control to the other stack control chain. However, this should not be relied on, as the frames would not be abandoned at that time if someone else happened to retain a pointer to the caller's control chain, but subsequently never returned to the frame held by the pointer. Cleanup forms are *not* run for frames abandoned when a stack pointer is released, either explicitly or by being garbage-collected. Cleanup forms are also not run for frames abandoned because of a control transfer via **ENVEVAL** or **ENVAPPLY**. Callers of **ENVEVAL** or **ENVAPPLY** should consider whether their intent would be served as well by **RETEVAL** or **RETAPPLY**, which *do* run cleanup forms in most cases.

Chapter 12 Miscellaneous

Section 12.4 System Version Information

All the functions listed on page 12.12 in the *Interlisp-D Reference Manual* have had their symbols moved to the LISP (CL) package. They are *not* shared with the INTERLISP package and any references to them in your code will need to be qualified i.e., CL:*name*.

Section 12.8 Pattern Matching

Pattern matching is no longer a standard part of the environment. The functionality for Pattern matching can be found in the Lisp Library Module called MATCH.

[This page intentionally left blank]

VOLUME II—ENVIRONMENT

Chapter 13 Interlisp Executive

[This chapter of the *Interlisp-D Reference Manual* has been renamed Chapter 13, Executives.]

Lisp has a new kind of Executive (or Exec), designed for use in an environment with both Interlisp and Common Lisp. This executive is available in three standard modes, distinguished by their default settings for package and readtable:

XCL — New Exec. Uses XCL readtable, XCL-USER package

CL — New Exec. Uses LISP readtable, USER package

IL — New Exec. Uses INTERLISP readtable, INTERLISP package

In addition, the old Interlisp executive, the "Programmer's Assistant", is still available in this release for the convenience of Koto users:

OLD-INTERLISP — Old "Programmer's Assistant" Exec. Uses OLD-INTERLISP-T readtable, INTERLISP package. It is likely that this executive will not be supported in future releases.

When Lisp starts, it is running a single executive, the XCL Exec. You can spawn additional executives by selecting EXEC from the background menu. The type of an executive is indicated in the title of its window; e.g., the initial executive has title "Exec (XCL)". Each executive runs in its own process; when you are finished with an executive, you can simply close its window, and the process is killed.

The new executive is modeled, somewhat, on the old "Programmer's Assistant" executive and, to a first approximation, you can type to it just as you did in past releases. You should note, however, that the default executive (XCL) expects Common Lisp input syntax, and reads symbols relative to the XCL-USER package. This means that to type Interlisp symbols, you must prefix the symbol with the characters "IL:" (in upper or lower case). And even in the new IL executive, the readtable being used is the new INTERLISP readtable, in which the characters colon (:), vertical bar (|) and hash (#) all have different meanings than in Koto.

The OLD-INTERLISP exec, with one exception, uses exactly the same input syntax as in Koto; this means in particular that colon cannot be used to type package-qualfied symbols, since colon is an ordinary character there. The one exception is that there *is* a package delimiter character in the OLD-INTERLISP readtable, should you have a need to use it—Control-↑, which usually echoes as "↑↑", though it may appear as a black rectangle in some fonts.

The new executive does differ from the old one in several respects, especially in terms of its programmatic interface. Complete details

of the new executive can be found in Appendix A. The Exec. Some of the important differences are:

- Executives are numbered

Executives, other than the first one, are labeled with a distinct number. This number appears in the exec window's title, and also in its prompt, next to the event number. The OLD-INTERLISP executive does not include this exec number.

- Event number allocation

The numbers for events are allocated at the time the prompt for the event is printed, but all execs still share a common event number space and history list. This means that ?? shows all events that have occurred in *any* executive, though not necessarily in the order in which the events actually occurred (since it is the order in which the event numbers were allocated). Events for which the type-in has not been completed are labeled "<in progress>" in the ?? listing. In the old executive, event numbers are not allocated until type-in is complete, which means that the number printed next to the prompt is not necessarily the number associated with the event, in the case that there has been activity in other executives.

In the new executive, relative event specifications are local to the exec; e.g., **-1** refers to the most recent event in that specific exec. In the old executive, **-1** referred to the immediately preceding event in *any* executive.

- New facility for commands

The old Executive has commands based on **LISPXMACROS**. The new Executive has its own command facility, **XCL:DEFCOMMAND**, which allows commands to be named without regard to package, and to be written with familiar Common Lisp style of argument list.

- Commands are typed *without* parentheses

In the old executive, a command could be typed with or without enclosing parentheses. In the new executive, a parenthesized form is always interpreted as an EVAL-style input, never a command.

- **SETQ** does not interact with the File Manager

In the Koto release, when you typed in the Exec

(**SETQ FOO** *some-new-value-for-FOO*)

the executive responded **(FOO reset)**, and the file package was told that **FOO**'s value changed. Any files on which **FOO** appeared as a variable would then be marked as needing to be cleaned up. If **FOO** appeared on no file, you'd be prompted to put it on one when you ran (**FILES?**).

This is still the case in the old executive. However, it is no longer the case in the new executive. 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** (rather than **VARS**), and it will be noticed by the File manager. If you want to change the value of the variable, you must either use

CL:DEFPARAMETER again, or edit the variable using **ED** (not **DV**).

- Programmatic interface completely different

As a first approximation, all the functions and variables in *IRM* Sections 13.3 (except the **LISPXPRINT** family) and 13.6 apply only to the Old Interlisp Executive, unless specified otherwise in Appendix A. In particular, the variables **PROMPT#FLG**, **PROMTPCHARFORMS**, **SYSPRETTYFLG**, **HISTORYSAVEFORMS**, **RESETFORMS**, **ARCHIVEFN**, **ARCHIVEFLG**, **LISPXUSERFN**, **LISPXMACROS**, **LISPXHISTORYMACROS** and **READBUF** are not used by the new Exec. The function **USEREXEC** invokes an old-style Executive, but uses the package and readtable of its caller. The function **LISPXUNREAD** has no effect on the new Exec. Callers of **LISPXEVAL** are encouraged to use **EXEC-EVAL** instead.

Some subsystems still use the old-style Executive—in particular, the tty structure editor.

Chapter 14 Errors and Breaks

Lisp extends the Interlisp break package to support multiple values and the Common Lisp lambda syntax. Interlisp errors have been converted to Common Lisp conditions.

Note that Sections 14.2 through 14.6 in the *Interlisp-D Reference Manual* have been replaced by new Debugger information (see *Common Lisp Implementation Notes)*.

Section 14.3 Break Commands

(II:14.6)

The **!EVAL** debugger command no longer exists.

(II:14.10-11)

The Break Commands = and -> are no longer supported.

Section 14.6 Creating Breaks with BREAK1

While the function **BREAK1** still exists, broken and traced functions are no longer redefined in terms of it. More primitive constructs are used.

Section 14.7 Signalling Errors

Interlisp errors now use the new XCL error system. Most of the functions still exist for compatibility with existing Interlisp code, but the underlying machinery is different. There are some incompatible differences, however, especially with respect to error numbers.

The old Interlisp error system had a set of registered error numbers for well known error conditions, and all other errors were identified

by a string (an error message). In the new Lisp error system, all errors are handled by signalling an object of type **XCL:CONDITION**. The mapping from Interlisp error numbers to Lisp conditions is given below in Section 14.10.

Since one cannot in general map a condition object to an Interlisp error number, the function **ERRORN** no longer exists. The equivalent functionality exists by examining the special variable ***LAST-CONDITION***, whose value is the condition object most recently signaled.

(ERRORX ERXM) calls **CL:ERROR** after first converting **ERXM** into a condition in the following way: 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; finally, if **ERXM** is already a condition, it is passed along unchanged. **ERRORX** also sets up a proceed case for **XCL:PROCEED**, which will attempt to re-evaluate the caller of **ERRORX**, much as OK did in the old Interlisp break package.

ERROR, **HELP**, **SHOULDNT**, **RESET**, **ERRORMESS**, **ERRORMESS1**, and **ERRORSTRING** work as before. All output is directed to ***ERROR-OUTPUT***, initially the terminal.

ERROR! is equivalent to the new error system's **XCL:ABORT** proceed function, except that if no **ERRORSET** or **XCL:CATCH-ABORT** is found, it unwinds all the way to the top of the process.

SETERRORN converts its arguments into a condition, then sets the value of ***LAST-CONDITION*** to the result.

Section 14.8 Catching Errors

ERRORSET, **ERSETQ** and **NLSETQ** have been reimplemented in terms of the new error system , but their behavior is essentially the same as before. **NLSETQ** catches all errors (conditions of type **CL:ERROR** and its descendants), and sets up a proceed case for **XCL:ABORT** so that **ERROR!** will return from it. **ERSETQ** also sets up a proceed case for **XCL:ABORT**, though it does not catch errors.

One consequence of the new implementation is that there are no longer frames named **ERRORSET** on the stack; programs that explicitly searched for such frames will have to be changed.

ERRORTYPELIST is no longer supported. The equivalent functionality is provided by default handlers. Although condition handlers provide a more powerful mechanism for programmatically responding to an error condition, old **ERRORTYPELST** entries generally cannot be translated directly. Condition handlers that want to resume a computation (rather than, say, abort from a well-know stack location) generally require the cooperation of a proceed case in the signalling code; there is no easy way to provide a substitute value for the "culprit" to be re-evaluated in a general way.

One important difference between **ERRORTYPELIST** and condition handlers is their behavior with respect to **NLSETQ**. In Koto, the relevant error handler on **ERRORTYPELST** would be tried, even for errors occurring underneath an **NLSETQ**. In Lyric, the **NLSETQ** traps all errors before the default condition handlers can see the

error. This means, for example, that the behavior of **(NLSETQ (OPENSTREAM --))** is now different if the **OPENSTREAM** causes a file not found error—in Koto, the system would search **DIRECTORIES** for the file; in Lyric, the **NLSETQ** returns **NIL** immediately without searching, since the default handler for **XCL:FILE-NOT-FOUND** is not invoked.

Section 14.9 Changing and Restoring System State

The special forms **RESETLST**, **RESETSAVE**, **RESETVAR**, **RESETVARS** and **RESETFORM** still exist, but are implemented by a new mechanism that also supports Common Lisp's **CL:UNWIND-PROTECT**. Common Lisp's **CL:THROW** and (in most cases) Interlisp's **RETFROM** and related control transfer constructs cause the cleanup forms of both **CL:UNWIND-PROTECT** and **RESETLST** (etc) to be performed. This is discussed in more detail in the notes for Chapter 11, the stack.

Section 14.10 Error List

Most of the Interlisp errors are mapped into condition types in Lisp. Some are no longer supported. Following is the list of error type mappings. The first name is the condition type that the error descriptor turns into. If there is a second name, it is the slot whose value is set to **CADR** of the error descriptor. Any additional pairs of items are the values of other slots set by the mapping. Attempting to use an unsupported error type number will result in a simple error to that effect.

0 Obsolete

1 Obsolete

2 **STACK-OVERFLOW**

3 **ILLEGAL-RETURN**

4 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'LIST**

5 **XCL:SIMPLE-DEVICE-ERROR** *MESSAGE*

6 **XCL:ATTEMPT-TO-CHANGE-CONSTANT**

7 **XCL:ATTEMPT-TO-RPLAC-NIL** *MESSAGE*

8 **ILLEGAL-GO TAG**

9 **XCL:FILE-WONT-OPEN PATHNAME**

10 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'CL:NUMBER**

11 **XCL:SYMBOL-NAME-TOO-LONG**

12 **XCL:SYMBOL-HT-FULL**

13 **XCL:STREAM-NOT-OPEN** *STREAM*

14 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'CL:SYMBOL**

15 Obsolete

16 **END-OF-FILE** *STREAM*

17 **INTERLISP-ERROR** *MESSAGE*

18 Not supported (control-B interrupt)

19 **ILLEGAL-STACK-ARG** *ARG*

20 Obsolete

21 **XCL:ARRAY-SPACE-FULL**

22 **XCL:FS-RESOURCES-EXCEEDED**

23 **XCL:FILE-NOT-FOUND** *PATHNAME*

24 Obsolete

25 **INVALID-ARGUMENT-LIST** *ARGUMENT*

26 **XCL:HASH-TABLE-FULL** *TABLE*

27 **INVALID-ARGUMENT-LIST** *ARGUMENT*

28 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'ARRAYP**

29 Obsolete

30 **STACK-POINTER-RELEASED** *NAME*

31 **XCL:STORAGE-EXHAUSTED**

32 Not supported (attempt to use item of incorrect type)

33 Not supported (illegal data type number)

34 **XCL:DATA-TYPES-EXHAUSTED**

35 **XCL:ATTEMPT-TO-CHANGE-CONSTANT**

36 Obsolete

37 Obsolete

38 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'READTABLEP**

39 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'TERMTABLEP**

40 Obsolete

41 **XCL:FS-PROTECTION-VIOLATION**

42 **XCL:INVALID-PATHNAME** *PATHNAME*

43 Not supported (user break)

44 **UNBOUND-VARIABLE** *NAME*

45 **UNDEFINED-CAR-OF-FORM** *FUNCTION*

46 **UNDEFINED-FUNCTION-IN-APPLY**

47 **XCL:CONTROL-E-INTERRUPT**

48 **XCL:FLOATING-UNDERFLOW**

49 **XCL:FLOATING-OVERFLOW**

50 Not supported (integer overflow)

51 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'CL:HASH-TABLE**

52 **TOO-MANY-ARGUMENTS** *CALLEE* **:MAXIMUM CL:CALL-ARGUMENTS-LIMIT**

Note that there are many other condition types in Lisp; see the error system documentation in the *Common Lisp Implementation Notes* for details.

Chapter 15 Breaking Functions and Debugging

In Lyric the uses of BREAK, TRACE, and ADVISE are unchanged, from the user's point of view, but the internals of their implementation are quite different.

For complete documentation on the new implementation of breaking, tracing and advising, see the *Common Lisp Implementation Notes*, Section 25.3.

In particular, you should note the following differences:

- The variable **BRKINFOLST** no longer exists and the format of the value of the variable **BROKENFNS** has changed. In addition, the **BRKINFO** property is no longer used.
- **BREAK** and **TRACE** no longer work on CLISP words.
- The **BREAKIN** and **UNBREAKIN** functions no longer exist. No comparable facility exists in Lisp. The user can manually insert calls to the Common Lisp function **CL:BREAK** in order to create a breakpoint at that point in the function.

Please note the following additional changes to breaking functions:

Section 15.1 Breaking Functions and Debugging

(**BREAK0** *FN WHEN COMS — —*) [Function]

The functio **BREAK0** now works when applied to an undefined function. This allows you to use the breaking facility to create "stubs" that generate a breakpoint when called. You can then examine the arguments passed and use the **RETURN** command in the debugger to return the proper result(s).

The "break commands" facility (the *COMS* argument) is no longer supported. **BREAK0** now signals an error when supplied with a non-**NIL** third argument. If you need finer control over the functioning of breakpoints you are directed to the **ADVISE** facility; it offers complete control of how and when the given function is evaluated.

Passing a non-atomic argument in the form **(***FN1* **IN** *FN2***)** as the first argument to **BREAK0** still has the effect of creating a breakpoint wherever *FN2* calls *FN1*. However, it no longer creates a function named *FN1***-IN-***FN2* to do so. In addition, the format of the value of the **NAMESCHANGED** property has changed and the **ALIAS** property is no longer used.

(**TRACE** *X*) [Function]

TRACE is no longer a special case of **BREAK**, though the functions **UNBREAK** and **REBREAK** continue to work on traced functions.

In addition, the function **TRACE** no longer calls **BREAK0** in order to do its job. Also, non-atomic arguments to **TRACE** no longer specify forms the user wishes to see in the tracing output.

(**UNBREAK** *X*) [Function]

The function **UNBREAK** is no longer implemented in terms of **UNBREAK0**, although that function continues to exist.

Section 15.2 Advising

The implementation of advising has been completely reworked. While the semantics implied by the code shown in Section 15.2.1 of the *Interlisp-D Reference Manual* is still supported, the details are quite different. In particular, it is now possible to advise functions that return multiple values and for **AFTER**-style advice to access those values. Also, all advice is now compiled, rather than interpreted. The advising facility no longer makes use of the special forms **ADV-PROG**, **ADV-RETURN**, and **ADV-SETQ**.

You should also note the following changes to the advise facility:

- The editing of advice has chnged slightly. In previous releases, the advice and original function-body were edited simultaneously. In Lyric, they can only be edited separately. When you finish editing the advice for a function, that function is automatically re-advised using the new advice.
- The variable **ADVINFOLST** no longer exists and the format of the value of the variable **ADVISEDFNS** has changed. In addition, the properties **ADVICE** and **READVICE** are no longer used, except in the handling of advice saved on files from previous releases. Advice saved in Lyric does not use the **READVICE** property.
- The function **ADVISEDUMP** no longer exists.
- Advice saved on files in previous releases can, in general, be loaded into the Lyric system compatibly. A known exception is the case in which a list of the form **(***FN1* **IN** *FN2***)** was given to the **ADVICE** or **ADVISE** file package commands. When **READVISE** is called on such a name, the old-style advice, on the **READVICE** property of the symbol *FN1***-IN-***FN2*, will not be found. This will eventually lead to an **XCL:ATTEMPT-TO-RPLAC-NIL** error. The user should evaluate the form
  (RETFROM 'READVISE1)
  in the debugger to proceed from the error and later evaluate
  (READVISE *FN1***-IN-***FN2***)**
  by hand to install the advice.

- The **ADVICE** and **ADVISE** File Manager commands now accept three kinds of arguments:
  a symbol, naming an advised function,
  a list in the form **(***FN1* **:IN** *FN2***)**, and
  a symbol of the form *FN1***-IN-***FN2*.
  Arguments of the form **(***FN1* **IN** *FN2***)** are not acceptable any longer. Arguments of the form *FN1***-IN-***FN2* should be converted into the equivalent form **(***FN1* **:IN** *FN2***)**.

(**ADVISE** *WHO WHEN WHERE WHAT*) [Function]

In the Lyric release of Lisp, **ADVISE** has some changes in the way arguments are treated and the possible values for those arguments. Most notably:

- In earlier releases, you could call **ADVISE** with only one argument, the name of a function. In this case, **AVISE** "set up" the named function for advising, but installed no advice. This usage is no longer supported.
- Previously, an undocumented value of **BIND** was accepted for the *WHEN* argument to **ADVISE**. This kind of advice is no longer supported. It can be adequately simulated using **AROUND** advice.

In addition, advising Common Lisp functions works somewhat differently with respect to a function's arguments. The arguments are not available by name. Instead, the variable **XCL:ARGLIST** is bound to a list of the values passed to the function and may be changed to affect what will be passed on.

As with the breaking facility (see above), **ADVISE** no longer creates a function named *FN1***-IN-***FN2* as a part of advising **(***FN1* **IN** *FN2***)**.

Chapter 16 List Structure Editor

The list structure editor, DEdit, is not part of the Lisp environment. It is now a Lisp Library Module. Chapter 16 has been renamed Structure Editor.

SEdit, the new Lisp editor, replaced DEdit in the Lyric release. The description of SEdit may be found in Appendix B of this volume. The commands used to invoke both SEdit and DEdit are the same.

Following is a description of the interface to the Lisp editor.

Switching Between Editors

If you have both SEdit and DEdit loaded, you can switch between them by calling: (**EDITMODE** *'EDITORNAME*) where *EDITORNAME* is one of the symbols SEdit or DEdit.

Packages

The **ED** editor interface accepts TYPE information from the Interlisp or Common Lisp packages.

Starting a Lisp Editor

In the XCL environment, calling ED with a pathname will start the editor on the coms of the file (as if DC had been called).

(ED *NAME &OPTIONAL OPTIONS*) [Function]

This function starts the Lisp editor. **ED** is the default interface to the editor. SEdit is the default Lisp editor. The same symbol, **ED**, is exported in both the IL and CL packages.

NAME is the name of any File Manager object.

OPTIONS is either a single symbol or a list of symbols, each of which is either a File Manager type or one or more of the keywords **:DISPLAY**, **:DONTWAIT**, **:CURRENT, :COMPILE-ON-COMPLETION, :CLOSE-ON-COMPLETION**, or **:NEW**. If exactly one File Manager type is given, **ED** tries to edit that type of definition for *NAME*. If more than one type is given in *OPTIONS*, **ED** will determine for which of them *NAME* has a definition. If a definition exists for more than one of the types, **ED** gives you a choice of which one to edit. If no File Manager types are given, **ED** treats *OPTIONS* as a list of all of the existing types; thus you are given a choice of all of the existing definitions of *NAME*.

The variable **FILEPKGTYPES** contains a complete list of the currently-known manager types.

If the keyword **:DISPLAY** is included in *OPTIONS*, **ED** uses menus for any prompting, (e.g., to choose one of several possible definitions to edit). If **:DISPLAY** is not included, **ED** prints its queries to and reads the user's replies from ***QUERY-IO*** (usually the Exec in which you are typing). Thus all of the following are correct ways to call the editor:

```
(ED 'NAME :DISPLAY)

(ED 'NAME 'FUNCTIONS)

(ED 'NAME '(:DISPLAY))

(ED 'NAME '(FUNCTIONS :DISPLAY))

(ED 'NAME '(FUNCTIONS VARIABLES :DISPLAY))
```

The other keywords are interpreted as follows:

:CURRENT

This is a new option with Medley that causes ED to call TYPESOF with SOURCE=CURRENT. This prevents TYPESOF from searching FILECOMS and from looking in WHERE-IS databases. The **CURRENT** option looks only for definitions that are currently loaded. When you know that the definition is loaded, use of the **CURRENT** option results in ED being significantly faster.

:DONTWAIT

Lets the edit interface return right away, rather than waiting for the edit to be complete. **DF**, **DV**, **DC**, and **DP** specify this option now, so editing from the exec will not cause the exec to wait.

:NEW

Lets you install a new definition for the name to be edited. You will be asked what type of dummy definition you wish to install based on which file manager types were included in *OPTIONS*.

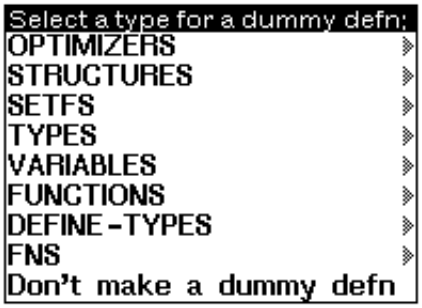
:COMPILE-ON-COMPLETION

This option specifies that the definition being edited should be compiled upon completion regardless of the completion command used.

:CLOSE-ON-COMPLETION

Tells the editor that it must close the editor window after the first completion. So in SEdit, CONTROL-X will close the window; shrinking the window is not allowed. Editor windows opened by the exec command **FIX** specify this option.

If *NAME* does not have a definition of any of the given types, **ED** can create a dummy definition of any of those types. If **:DISPLAY** is provided in *OPTIONS,* **ED** will pop-up the following menu asking you which type of definition to install. Select the template for the type of definition you wish to create from the DEFN menus and submenus:



New kinds of dummy definitions can be added to the system through the use of the **:PROTOTYPE** option to **XCL:DEFDEFINER.**

Mapping the Old Edit Interface to ED

The old functions for starting the Lisp editor (**DF**, **DV**, **DP**, and **DC**) have been reimplemented so that they work with Common Lisp. The old edit commands map to the new editor function (ED) as follows:

DF *NAME* ⇒ (ED '*NAME* '(FUNCTIONS FNS :DONTWAIT))

DV *NAME* ⇒ (ED '*NAME* '(VARIABLES VARS :DONTWAIT))

DP *NAME* ⇒ (ED '*NAME* '(PROPERTY-LIST :DONTWAIT))

DP *NAME MYPROP* ⇒ (ED '(*NAME MYPROP*) '(PROPS :DONTWAIT))

DC NAME ⇒ (ED 'NAME '(FILES :DONTWAIT))

Thus, for example, when **DF** is invoked it looks first for Common Lisp `FUNCTIONS` and then for Interlisp `FNS`. **DV**, **DP** and **DC** operate in an analogous fashion.

Editing Values Directly

The TYPE you specify for the object you want to edit determines how that object is edited, i.e. by DEFINITION or VALUE. Normally you want to edit the DEFINITION (this is the default). For example, suppose *FOO* is defined as a variable; to start the editor on the DEFINITION of *FOO,* use the form:

```
(ED 'FOO)   or  (ED 'FOO 'VARIABLES)
```

There may be times when you do not have access to the DEFINITION of an object that you need to edit. This can occur when you do not have the source code loaded. You can edit its VALUE directly using the form:

FOR VARIABLES: ⇒ (ED '*NAME* 'IL:VARS)

FOR FUNCTIONS: ⇒ (ED '*NAME* 'IL:FNS)

By starting the editor on the VALUE of an object, you can change its value without changing its definition. (AR 8971)

To start the editor on the VALUE of *FOO*, for example, use the form:

```
(ED 'FOO 'VARS)
```

EXAMPLE:

When you load a compiled file, the DEFINITION of an object is not loaded. Only the VALUE is loaded. The compiler does not store the defining forms for objects. Suppose you have compiled code for a system file loaded, but you do not have access to the sources that contain the DEFINITIONS, and you need to change the value of a system variable, say NETWORKLOGINFO. This variable has a defining form and the system knows this, but the form is not loaded and is not available. You can edit the VALUE of the variable directly using:

```
(ED 'NETWORKLOGINFO 'IL:VARS)
```

An editor window opens displaying the VALUE of NETWORKLOGINFO:

```
SEdit NETWORKLOGINFO Package: INTERLISP
((TENEX (LOGIN "LOGIN " USERNAME " " PASSWORD " ↑M")
        (ATTACH "ATTACH " USERNAME " " PASSWORD " ↑M")
        (WHERE "WHERE " USERNAME CR
               "ATTACH " USERNAME
               " " PASSWORD CR))
 (TOPS20 (LOGIN "LOGIN " USERNAME CR PASSWORD CR)
         (ATTACH "ATTACH " USERNAME "lama " CR PASSWORD CR)
         (WHERE "LOGIN " USERNAME CR PASSWORD CR))
 (UNIX (LOGIN WAIT CR WAIT USERNAME CR WAIT PASSWORD CR))
 (IFS (LOGIN "Login " USERNAME " " PASSWORD CR) (ATTACH))
 (NS (LOGIN "Logon" CR USERNAME CR PASSWORD CR))
 (VMS (LOGIN USERNAME CR PASSWORD CR)))
```

Section 16.18 Editor Functions

(II:16.74)

The function **FINDCALLERS** has the following limitations in Lisp:

1. **FINDCALLERS** only identifies by name the occurrences inside of Interlisp FNS, not Common Lisp FUNCTIONS.

2. Because **FINDCALLERS** uses a textual search, it may report more occurrences of the specified *ATOMS* than there actually are, if the file contains symbols by the same name in another package, or symbols with the same p-name but different alphabetic case. **EDITCALLERS** still edits only the actual occurrences, since it reads the functions and operates on the real Lisp structure, not its printed representation.

Chapter 17 File Package

The Interlisp-D File Package has been renamed the File Manager. Its operation is unchanged; however, it has been extended to manipulate, load and save Common Lisp functions, variables, etc. It also allows specification of the reader environment (package and readtable) to use when writing and reading a file, solving the problem of compatibility between old and new (Common Lisp) syntax.

Note that although source files from earlier releases can be loaded into Lyric, files produced by the File Manager in the Lyric release cannot be loaded into previous releases. This is true for several reasons, the most important being that previous releases did not have packages, so symbols cannot be read back consistently.

The new File Manager includes several new types to deal with the various definition forms supported in Xerox Common Lisp. The following table associates each new type with the forms that produce definitions of that type:

FUNCTIONS	**CL:DEFUN, CL:DEFMACRO, CL:DEFINE-MODIFY-MACRO, XCL:DEFINLINE, XCL:DEFDEFINER, XCL:DEFINE-PROCEED-FUNCTION.**
VARIABLES	**CL:DEFCONSTANT, CL:DEFVAR, CL:DEFPARAMETER, XCL:DEFGLOBALVAR, XCL:DEFGLOBALPARAMETER**
STRUCTURES	**CL:DEFSTRUCT, XCL:DEFINE-CONDITION**
TYPES	**CL:DEFTYPE**
SETFS	**CL:DEFSETF, CL:DEFINE-SETF-METHOD**
DEFINE-TYPES	**XCL:DEF-DEFINE-TYPE**
OPTIMIZERS	**XCL:DEFOPTIMIZER**
COMMANDS	**XCL:DEFCOMMAND**

Note that the types listed above, as well as all the old File Manager types, are symbols in the INTERLISP package. In addition, the "filecoms" variable of a file and its rootname are also both in the INTERLISP package. You should be careful when typing to a Common Lisp exec to qualify all such symbols with the prefix **IL:**; e.g.,

3>**(setq il:foocoms '((il:functions bar) (il:prop il:filetype il:foo)))**

to indicate you want the function BAR (in the current package) to live on a file with rootname FOO, and also that FOO's FILETYPE property should be saved.

Reader Environments and the File Manager

(II:17.1)

In order for **READ** to correctly read back the same expression that **PRINT** printed, it is necessary that both operations be performed in the same reader environment, i.e., the collection of parameters that affect the way the reader interprets the characters appearing on the input stream. In previous releases of Interlisp there was, for all practical purposes, a single such environment, defined entirely by the readtable *FILERDTBL*. In the Lyric release of Lisp there are two significantly different readtables in which to read (Common Lisp and Interlisp). In addition, there are more parameters than just the readtable that can potentially affect **READ:** the current package and the read base (the bindings of ***PACKAGE*** and ***READ-BASE***).

To handle this diversity, a new type of object is introduced, the **READER-ENVIRONMENT**, consisting of a readtable, a package, and a read/print base. Every file produced by the File Manager has a header at the beginning specifying the reader environment for that file. **MAKEFILE** and the compiler produce this header, while **LOAD**, **LOADFNS**, and other file-reading functions read the header in order to set their reading environment correctly. Files written in older releases of Lisp lack this header and are interpreted as

having been written in the environment consisting of the readtable *FILERDTBL* and the package INTERLISP. Thus, you need take no special action to be able to load Koto source files into Lyric; characters that are "special" in Common Lisp, such as colon, semi-colon and hash, are interpreted as the "ordinary" characters they were in Koto.

The File Manager's reader environments are specified as a property list of alternating keywords and values of the form (**:READTABLE** *readtable* **:PACKAGE** *package* **:BASE** *base*). The **:BASE** pair is optional and defaults to 10. The values for *readtable* and *package* should either be strings naming a readtable and package, or expressions that can be evaluated to produce a readtable and package. In the former case, the readtable or package *must* be one that already exists in a virgin Lisp sysout (or at least in any Lisp image in which you might attempt *any* operation that reads the file). If an expression is used, care should be exercised that the expression can be evaluated in an environment where no packages or readtables, other than the documented ones, are presumed to exist. For hints and guidelines on writing the *package* expression for files that create or use their own private packages, please see Chapter 11 of the *Common Lisp Implementation Notes*.

When **MAKEFILE** is writing a source file, it uses the following algorithm to determine the reading environment for the new file:

1. If the root name for the file has the property **MAKEFILE-ENVIRONMENT**, the property's value is used. It should be in the form described above. Note that if you want the file always to be written in this environment, you should save the **MAKEFILE-ENVIRONMENT** property itself on the file, using a (**PROP MAKEFILE-ENVIRONMENT** *file*) command in the filecoms.

2. If a previous version of the file exists, **MAKEFILE** uses the previous version's environment. **MAKEFILE** does this even when given option **NEW** or the previous version is no longer accessible, assuming it still has the previous version's environment in its cache. If the previous version was written in an older release, and hence has no explicit reader environment, **MAKEFILE** uses the environment (**:READTABLE** "INTERLISP" **:PACKAGE** "INTERLISP" **:BASE** 10).

3. If no previous version exists (this is a new file), **MAKEFILE** uses the value of ***DEFAULT-MAKEFILE-ENVIRONMENT***, initially **(:READTABLE** "XCL" **:PACKAGE** "INTERLISP" **:BASE** 10).

Note that changing the value of ***DEFAULT-MAKEFILE-ENVIRONMENT*** only affects new files. If you decide you don't like the environment in which an existing file is written, you must give the file a **MAKEFILE-ENVIRONMENT** property to override any prior default.

Since the XCL readtable is case-insensitive, you should avoid using it for files that contain many mixed-case symbols or old-style Interlisp comments, as these will be printed with many escape

delimiters. This is why the default for reprinted Koto sources is the INTERLISP readtable.

The readtable named LISP (the pure Common Lisp readtable) should ordinarily not be used as part of a **MAKEFILE** environment. It exists solely for the use of "pure" Common Lisp (as in the CL Exec), and thus has no provision for font escapes (inserted by the Lisp prettyprinter) to be treated as whitespace. Most users will want to use either XCL or INTERLISP as the readtable for files.

If the environment for the new version of the file differs from that of the previous version, **MAKEFILE** copies unchanged FNS definitions by actually reading from the old file, rather than just copying characters as it otherwise would. Similarly, when **RECOMPILE** or **BRECOMPILE** attempt to recompile a file for which the previous compiled version's reader environment is different, they must compile afresh all the functions on the file, i.e., they behave like **TCOMPL** or **BCOMPL**.

Modifying Standard Readtables

In the past, programmers have been periodically tempted to change standard readtables, such as **T** and **FILERDTBL**, typically by adding macros to read certain objects in a convenient way. For example, the PQUOTE LispUsers module defined single quote as a macro in **FILERDTBL**. Unfortunately, changing a standard readtable means that unless you are very careful, you cannot read other users' files that were not written with your change, and they cannot read your files without obtaining your macro. Furthermore, the effects are often subtle. Rather than breaking, the system merely reads the file incorrectly. For example, reading a file written with PQUOTE in an environment lacking PQUOTE produces many symbols with a single quote packed on the front.

This confusion can be avoided with MAKEFILE reader environments. To add your own special macro:

1. Copy some standard readtable; e.g., (COPYRDTBL "INTERLISP").
2. Give it a distinguished name of its own, by using (READTABLEPROP *rdtbl* 'NAME "*yourname*").
3. Make your change in the copied readtable.
4. Use your new private readtable to write your files: use its name ("*yourname*") in the **MAKEFILE-ENVIRONMENT** property of selected files and/or change ***DEFAULT-MAKEFILE-ENVIRONMENT*** to affect all your new files.
5. Make sure to save your new readtable. It is usually most convenient to include the code to create it (steps 1-3) in your system initialization, but you could even write a self-contained expression to use in a single file's **MAKEFILE-ENVIRONMENT** property.

With this strategy, your system will read all files in the proper environment—your own files with your private readtable and other users' files in their environments, including the standard environments, which you have carefully avoided polluting. If

another user tries to load one of your files into an environment that doesn't know about your private readtable, **LOAD** will give an error immediately (readtable not found), rather than loading the file quietly but incorrectly.

Programmer's Interface to Reader Environments

The following function and macro are available for programmers to use. Note that reader environments only control the parameters that determine read/print consistency. There are other parameters, such as ***PRINT-CASE***, that affect the appearance of the output without affecting its ability to be read. Thus, reader environments are not sufficient to handle problems of, for example, repainting expressions on the display in exactly the same total environment in which they were first written.

(**MAKE-READER-ENVIRONMENT** *PACKAGE READTABLE BASE*) [Function]

Creates a **READER-ENVIRONMENT** object with the indicated components. The arguments must be valid values for the variables ***PACKAGE***, ***READTABLE*** and ***PRINT-BASE***; names are not sufficient. If any of the arguments is **NIL**, the current value of the corresponding variable is used. Thus **(MAKE-READER-ENVIRONMENT)** returns an object that captures the current environment.

(**WITH-READER-ENVIRONMENT** *ENVIRONMENT . FORMS*) [Macro]

Evaluates each of the *FORMS* with ***PACKAGE***, ***READTABLE***, ***PRINT-BASE*** and ***READ-BASE*** bound to the values in the *ENVIRONMENT* object. Both ***PRINT-BASE*** and ***READ-BASE*** are bound to the single *BASE* value in the environment.

(**GET-ENVIRONMENT-AND-FILEMAP** *STREAM DONTCACHE*) [Function]

Parses the header of a file produced by the File Manager and returns up to four values:

1. The reader environment in which the file was written;
2. The file's "filemap", used to locate functions on the file;
3. The file position where the FILECREATED expression starts; and
4. A value used internally by the File Manager.

STREAM can be a full file name, in which case this function returns NIL unless the information was previously cached. Otherwise, *STREAM* is a stream open for input on the file. It must be randomly accessible (unless information is available from the cache). If the file is in Common Lisp format (it begins with a comment), then value 1 is the default Common Lisp reader environment (readtable LISP, package USER) and the other values are NIL. Otherwise, if the file is not in File Manager format, values 1 and 2 are NIL, 3 is zero.

If *DONTCACHE* is true, the function does not cache any information it learns about File Manager files; otherwise, the information is cached to speed up future inquiries.

Section 17.1 Loading Files

(II:17.5)

Integration of Interlisp and Common Lisp LOAD functions

There are four kinds of files that can be loaded in Lisp:

1. Interlisp and Common Lisp source files produced by the File Manager using, for example, the **MAKEFILE** function.
2. Standard Common Lisp source files produced with a text editor either in Lisp or from some other Common Lisp implementation.
3. DFASL files of compiled code, produced by the new XCL Compiler, **CL:COMPILE-FILE** (extension DFASL)
4. LCOM files of compiled code, produced by the old Interlisp Compiler (**BCOMPL, TCOMPL**).

Types 1 and 4 were the only kind of files that you could load in Koto; types 2 and 3 are new with Lyric. Both **IL:LOAD** and **CL:LOAD** are capable of loading all four kinds of files. However, they use the following rules to make the types of files unambiguous so that they can be loaded in the correct reader environment.

- If the file begins with an open parenthesis (possibly after whitespace and font switch characters), it is assumed to be of type 1 or 4: files produced by the File Manager. The first expression on the file (at least) is assumed to be written in the old **FILERDTBL** environment; for new Lyric files this expression defines the reader environment for the remainder of the file. See the section, Reader Environments and File Manager for details.
- If the file begins with the special FASL signature byte (octal 221), it is assumed to be a compiled file in FASL format, and is processed by the FASL loader. The FASL loader ignores the *LDFLG* argument to **IL:LOAD**, treating all files as though *LDFLG* were **SYSLOAD** (redefinition occurs, is not undoable, and no File Manager information is saved).
- If the file begins with a semicolon, it is assumed to be a pure Common Lisp file. The expressions on the file are read with the standard Common Lisp readtable and in package USER (unless a package argument was given to **LOAD**; see below).
- If the file begins with any other character, **LOAD** doesn't know what to do. Currently, it treats the file as a pure Common Lisp file (as if it started with a comment).

Thus, if you prepare Common Lisp text files you should be sure to begin them with a comment so that **LOAD** can tell the file is in Common Lisp syntax.

The function **CL:LOAD** accepts an additional keyword **:PACKAGE**, whose value must be a package object; the function **IL:LOAD** similarly has an optional fourth argument *PACKAGE*. If a package argument is given, then **LOAD** reads Common Lisp text files (type 2 above) with ***PACKAGE*** bound to the specified package. In the

case of File Manager files (types 1 and 4), the package argument overrides the package specified in the file's reader environment.

(II:17.6-17.8)

The Interlisp functions **LOADFNS**, **LOADFROM**, **LOADVARS** and **LOADCOMP** do not work on FASL files. They do still work on files produced by the old compiler (extension LCOM).

(II:17.9)

FILESLOAD (also used by the File Manager's **FILES** command) now searches for compiled files by looking for a file by the specified name whose extension is in the list ***COMPILED-EXTENSIONS***. The default value for ***COMPILED-EXTENSIONS*** in the Lyric release is (DFASL LCOM). It searches the list of extensions in order for each directory on the search path. This means that FASL files are loaded in preference to old-style compiled files.

Section 17.2 Storing Files

The Lyric release contains two different compilers, the Interlisp Compiler that was present in Koto and previous releases, and the new XCL Compiler (see the next section, Chapter 18 Compiler). With more than one compiler available, the question arises as to which compiler will be used by the functions **CLEANUP** and **MAKEFILE.** The default behavior of these functions in Lyric is to always use the new XCL Compiler. This default can be changed, either on a file-by-file basis or system-wide. Most users, however, will have no need to change the default.

When the *C* or *RC* option has been given to **MAKEFILE,** the system first looks for the value of the **FILETYPE** property on the symbol naming the file. For example, for the file "{DSK}<LISPFILES>MYFILE", the property list of the symbol **MYFILE** would be examined.

The **FILETYPE** property should be either a symbol from the list below or a list containing one of those symbols. The following symbols are allowed and have the given meanings:

:TCOMPL Compile this file by calling either **TCOMPL** or **RECOMPILE**, depending upon which of the C or RC options was passed to **MAKEFILE**.

:BCOMPL Compile this file by calling either **BCOMPL** or **BRECOMPILE**, depending upon which of the C or RC options was passed to **MAKEFILE**. This is equivalent to the Koto behavior.

:COMPILE-FILE Compile this file by calling **CL:COMPILE-FILE**, regardless of which option was passed to **MAKEFILE**.

If no **FILETYPE** property is found, then the function whose name is the value of the variable ***DEFAULT-CLEANUP-COMPILER*** is used. The only legal values for this variable are **TCOMPL**, **BCOMPL**, and **CL:COMPILE-FILE**. Initially, ***DEFAULT-CLEANUP-COMPILER*** is set to **CL:COMPILE-FILE**.

If you choose to set the **FILETYPE** property of file name, you should take care that the filecoms for that file saves the value of that property on the file. This will ensure that the same compiler

will be used every time the file is loaded. To save the value of the property, you should include a line in the coms like the following:

```
(PROP FILETYPE MYFILE)
```

where MYFILE is the symbol naming your file.

Section 17.8.2 Defining New File Manager Types

(II:17.30)

The File Manager has been extended to allow File Manager types that accept any Lisp object as a name. A consequence of this is that any user-defined type's **HASDEF** function should be prepared to accept objects other than symbols as the *NAME* argument. Names are compared using **EQUAL**.

Definers: A New Facility for Extending the File Manager

The Definer facility is provided to make the process of adding a certain common kind of File Manager type easy. All of the new File Manager types in the Lyric release (including **FUNCTIONS**, **VARIABLES**, **STRUCTURES**, etc.) and almost all of the new defining macros (including **CL:DEFUN**, **CL:DEFPARAMETER**, **CL:DEFSTRUCT**, etc.) were themselves created using the Definer facility.

In previous releases, adding new types and commands to the File Manager involved deeply understanding the way in which it worked and defining a number of functions to carry out certain operations on the new type/command. Further, making functions and macros save away definitions of the new type was similarly subtle and generally difficult or complicated to do. With the addition of Common Lisp, it was realized that a large number of new types and commands would be added, all needing essentially the same implementation of the various operations. In addition, many new defining macros were to be added and all of them needed to save definitions.

As an explanation of the Definer facility, we will describe how **VARIABLES** and **CL:DEFPARAMETER** could be added into the system, if they were not already there.

First, a little background about our example. The macro **CL:DEFPARAMETER** is used in Common Lisp to globally declare a given variable to be special and to give it an initial value. (For the purposes of this example, we will ignore the documentation-string given to real **CL:DEFPARAMETER** forms.) The value of a call to the macro should be the name of the variable being defined. An acceptable definition of this macro might appear as follows:

```
(DEFMACRO CL:DEFPARAMETER (SYMBOL EXPRESSION)
   `(PROGN
      (CL:PROCLAIM '(CL:SPECIAL ,SYMBOL))
      (SETQ ,SYMBOL ,EXPRESSION)
      ',SYMBOL))
```

There are some problems with using such a simple definition in the Lisp environment, however. For example, if a call to this macro were typed to the Exec, the File Manager would not be told to notice it. Thus, there would be no convenient way to remember to

add the form to the filecoms of some file and thus to save it away. Also, note that the macro does not pay attention to the **DFNFLG** variable; thus, loading a file containing a **CL:DEFPARAMETER** form would always set the variable to the value of the initial expression, even when **DFNFLG** was set to **ALLPROP**. This could make editing code using this variable difficult.

We will now proceed to fix these problems by getting the Definer facility involved. There are two steps involved in using Definers:

- Unless one of the currently-existing File Manager types is appropriate for definitions using the new macro, a new type must be created. The macro **XCL:DEF-DEFINE-TYPE** is used for this purpose.
- The macro must be defined in such a way that the File Manager can tell that it should notice and save uses of the macro and under which File Manager type the uses should be saved. The macro **XCL:DEFDEFINER** is used for this purpose.

Since we are pretending for the example that the File Manager type **VARIABLES** is not defined, we decide that definitions using **CL:DEFPARAMETER** should not be given any of the already-existing types. We must define a type, therefore, and we decide to call it **VARIABLES**. The following **XCL:DEF-DEFINE-TYPE** form will do the trick:

```
(XCL:DEF-DEFINE-TYPE VARIABLES "Common Lisp
variables")
```

The first argument to **XCL:DEF-DEFINE-TYPE** is the name for the new type. The second argument is a descriptive string, to be used when printing out messages about the type.

With the new type thus created, we can now use **XCL:DEFDEFINER** to redefine the macro. Simply changing the word **DEFMACRO** into **XCL:DEFDEFINER** and adding an argument specifying the new type suffices to change our earlier definition into a use of the Definer facility:

```
(XCL:DEFDEFINER CL:DEFPARAMETER VARIABLES
                                    (SYMBOL EXPRESSION)
   `(PROGN
      (CL:PROCLAIM '(CL:SPECIAL ,SYMBOL))
      (SETQ ,SYMBOL ,EXPRESSION)
      ',SYMBOL))
```

(In fact, we could also remove the final **',SYMBOL**; **XCL:DEFDEFINER** automatically arranges for the new macro to return the name of the new definition.) Now, if we were to type the form

```
(CL:DEFPARAMETER *FOO* 17)
```

into the Exec and then call the function **FILES?**, we would be presented with something like the following:

```
24> (FILES?)
the Common Lisp variables: *FOO*
...to be dumped.  want to say where the above go?
```

As with other File Manager types, our definitions are being kept track of. If we answer Yes to the above question and specify a file

in which to save the definition, a command like the following will be added to the filecoms:

```
(VARIABLES *FOO*)
```

Actually, the output from **FILES?** as shown above is not quite accurate. In reality, we would also be asked about the following:

```
the Common Lisp functions/macros: CL:DEFPARAMETER
the Definition types: VARIABLES
```

The File Manager is also watching for new types and new Definers being created and will let us save those definitions as well. These would be listed in the filecoms as follows:

```
(DEFINE-TYPES VARIABLES)
(FUNCTIONS CL:DEFPARAMETER)
```

All of these definitions are full-fledged File Manager citizens. The functions **GETDEF**, **HASDEF**, **PUTDEF**, **DELDEF**, etc. all work with the new type. We can edit the definition of ***FOO*** above simply by specifying the type to the **ED** function:

```
(ED '*FOO* 'VARIABLES)
```

When we exit the editor, the new definition will be saved and, unless **DFNFLG** is set to **PROP** or **ALLPROP**, evaluated.

It is now time to fully describe the macros **XCL:DEF-DEFINE-TYPE** and **XCL:DEFDEFINER**.

XCL:DEF-DEFINE-TYPE *NAME DESCRIPTION* &KEY :UNDEFINER [*Macro*]

Creates a new File Manager type and command with the given *NAME*. The string *DESCRIPTION* will be used to describe the type in printed messages. The new type implements **PUTDEF** operations by evaluating the definition form, **GETDEF** and **HASDEF** by looking up the given name in an internal hash-table, using **EQUAL** as the equality test on names, and **DELDEF** by removing any named definition from the hash-table. If the **:UNDEFINER** argument is provided, it should be the name of a function to be called with the *NAME* argument to any **DELDEF** operations on this type. The **:UNDEFINER** function can perform any other operations necessary to completely delete a definition.

XCL:DEF-DEFINE-TYPE forms are File Manager definitions of type **DEFINE-TYPES**.

As an example of the full use of **XCL:DEF-DEFINE-TYPE**, here is the complete definition of the type VARIABLES as it exists in the Lyric release:

```
(XCL:DEF-DEFINE-TYPE  VARIABLES "Common Lisp variables"
                      :UNDEFINER UNDOABLY-MAKUNBOUND)
```

The function **UNDOABLY-MAKUNBOUND** is described in Appendix D of these Release Notes.

XCL:DEFDEFINER {*NAME* | (*NAME* {*OPTION*}*)} *TYPE ARG-LIST* &BODY *BODY* [*Macro*]

Creates a macro named *NAME*, calls to which are seen as File Manager definitions of type *TYPE*. *TYPE* must be a File Manager type previously defined using **XCL:DEF-DEFINE-TYPE**. *ARG-LIST* and *BODY* are precisely as in **DEFMACRO**. A macro defined using

XCL:DEFDEFINER differs from one defined using **DEFMACRO** in the following ways:

- *BODY* will be evaluated if and only if the value of **DFNFLG** is not one of **PROP** or **ALLPROP**.
- The form returned by *BODY* will be evaluated in a context in which the File Manager has been temporarily disabled. This allows Definers to expand into other Definers without the subordinate ones being noticed by the File Manager.
- Calls to Definers return the name of the new definition (as, for example, **CL:DEFUN** and **CL:DEFPARAMETER** are defined to do).
- Calls to Definers are noticed and remembered by the File Manager, saved as a definition of type *TYPE*.
- SEdit- and Interlisp-style comment forms (those with a CAR of IL:*) are stripped from the macro call before it is passed to *BODY*. (This comment-removal is partially controlled by the value of the variable ***REMOVE-INTERLISP-COMMENTS***, described below.)

The following *OPTION*s are allowed:

(:UNDEFINER *FN*)

If **DELDEF** is called on a name whose definition is a call to this Definer, *FN* will be called with one argument, the name of the definition. This option allows for Definer-specific actions to be taken at **DELDEF** time. This is useful when more than one Definer exists for a given type. *FN* should be a form acceptable as the argument to the **FUNCTION** special form.

(:NAME *NAME-FN*)

By default, the Definer facility assumes that the first argument to any macro defined using **XCL:DEFDEFINER** will be the name under which the definition should be saved. This assumption holds true for almost all Common Lisp defining macros, including **CL:DEFUN**, **CL:DEFMACRO**, **CL:DEFPARAMETER** and **CL:DEFVAR**. It doesn't work, however, for a few other forms, such as **CL:DEFSTRUCT** and **XCL:DEFDEFINER** itself. When defining a macro for which that assumption is false, the **:NAME** option should be used. *NAME-FN* should be a function of one argument, a call to the Definer. It should return the Lisp object naming the given definition (most commonly a symbol, but any Lisp object is permissible). For example, the **:NAME** option in the definitions of **CL:DEFSTRUCT** and **XCL:DEFDEFINER** is as follows:

```
(:NAME (LAMBDA (FORM)
          (LET ((NAME (CADR FORM)))
             (COND ((LITATOM NAME)
                    NAME)
                   (T  (CAR NAME))))))
```

NAME-FN should be a form acceptable as the argument to the **FUNCTION** special form (i.e., a symbol naming a function or a LAMBDA-form).

(:PROTOTYPE *DEFN-FN*)

When the editor function **ED** is passed a name with no definitions, the user is offered a choice of several ways to create a prototype definition. Those choices are specified with the **:PROTOTYPE** option to **XCL:DEFDEFINER**. *DEFN-FN* should be a function of one argument, the name to be defined using this Definer. *DEFN-FN* should return either NIL, if no definition of that name can be created with this Definer, or a form that, when evalauted, would create a definition of that name. For example, the **:PROTOTYPE** option for **CL:DEFPARAMETER** might look as follows:

```
(:PROTOTYPE (LAMBDA (NAME)
                (AND (LITATOM NAME)
                     '(CL:DEFPARAMETER ,NAME "Value"))))
```

An example using all of the features of **XCL:DEFDEFINER** is the definition of **XCL:DEFDEFINER** itself, which begins as follows:

```
(XCL:DEFDEFINER (XCL:DEFDEFINER
                    (:UNDEFINER \DELETE-DEFINER)
                    (:NAME
                       (LAMBDA (FORM)
                          (LET ((NAME (CADR FORM)))
                             (COND ((LITATOM NAME)
                                    NAME)
                                   (T  (CAR NAME))))))
                    (:PROTOTYPE
                       (LAMBDA (NAME)
                          (AND (LITATOM NAME)
                               '(XCL:DEFDEFINER ,NAME "Type"
                                                ("Arg List")
                                                "Body")))))
                FUNCTIONS
                (NAME-AND-OPTIONS TYPE ARG-LIST &BODY BODY)
    ...)
```

The following variable is used in the process of removing SEdit- and Interlisp-style comments from Definer forms:

REMOVE-INTERLISP-COMMENTS [*Variable*]

Interlisp-style comments are forms whose **CAR** is the symbol **IL:***. It is possible for certain lists in Lisp code to begin with **IL:*** but not be a comment (for example, a **SELECTQ** clause). When such a list is discovered, the value of ***REMOVE-INTERLISP-COMMENTS*** is examined. If it is **T**, the list is assumed to be a comment and is removed without comment. If it is **:WARN**, a warning message is printed, saying that a possible comment was not stripped from the code. If ***REMOVE-INTERLISP-COMMENTS*** is **NIL**, the list is not removed, but no warning is printed. This variable is initially set to **:WARN**.

(CL:EVAL-WHEN *WHEN COM ... COM)* [*File Package Command*]

Interprets each of the commands *COM ... COM* as a file package command, but output is wrapped in CL:EVAL-WHEN.

EXAMPLE:

```
(CL:EVAL-WHEN (CL:EVAL CL:COMPILE)
  (OPTIMIZERS FOO))
```

will cause the following to be written to the file:

```
(CL:EVAL-WHEN (CL:COMPILE)
 (DEFOPTIMIZER FOO <optimizer for FOO>))
```

Chapter 18 Compiler

The Lyric release contains two distinct Lisp compilers:

- The Interlisp Compiler, described in detail in Section 18 of the *IRM,*
- The new XCL Compiler, described in the *Common Lisp Implementation Notes*.

The Interlisp Compiler provides compatibility with previous releases of Interlisp-D. It continues to work in very much the same way as it did in Koto; as before, it compiles all of the Interlisp language. The Interlisp Compiler does not, however, compile the Common Lisp language and will not be extended to do so. The Lyric release is the last release to contain the Interlisp Compiler as a component; future releases will have only the new XCL Compiler. The XCL Compiler is designed to handle both Interlisp and Common Lisp.

Several incompatible changes have been made in the compiled object code produced by the Interlisp Compiler. This means that *all user code must be recompiled in Lyric*. Code compiled in Koto or previous releases will not load into Lyric, and code compiled in Lyric wil not load into earlier releases. The filename extension for Interlisp compiled files has been changed from DCOM to LCOM in order to minimize possible confusion.

The XCL Compiler writes its output on a new kind of object file, the DFASL file. These files are quite different from the DCOM/LCOM files produced by the Interlisp Compiler. DFASL files are somewhat more compact, much faster to load and can represent a wider range of data objects than was possible in LCOMs.

Interlisp source files from Koto can be compiled using the new XCL compiler. However, some files need to be remade in Lyric before compilation: files containing bitmaps, Interlisp arrays, or the **UGLYVARS** and/or **HORRIBLEVARS** File Manager commands. To compile such a file, first **LOAD** it, then call **MAKEFILE** to write it back out. This action causes the bitmaps and other unusual objects to be written back in a format acceptable to the new compiler.

The default behavior of the File Manager's **CLEANUP** and **MAKEFILE** functions is to use the new XCL Compiler to compile files, rather than the old Interlisp Compiler. To change this behavior, see Section 17.2, Storing Files.

Note that if you call the compiler explicitly, rather than via **CLEANUP** or **MAKEFILE**, you should be careful to specify the correct compiler. The new compiler is invoked by calling **CL:COMPILE-FILE**. If you inadvertantly call **BCOMPL** on a file for which **CLEANUP** has routinely been using the new XCL compiler, there are two undesirable consequences: (1) Any Common Lisp

functions on the file will not be compiled (the Interlisp compiler does not recognize **CL:DEFUN**), and (2) the DFASL files produced by earlier calls on the XCL compiler will still be loaded by **FILESLOAD** in preference to the LCOM file produced by **BCOMPL**.

Lisp provides a facility, **XCL:DEFOPTIMIZER**, by which you can advise the compiler about efficient compilation of certain functions and macros. **XCL:DEFOPTIMIZER** works with both the old Interlisp Compiler and the Lyric XCL Compiler. See the *Common Lisp Implementation Notes* for a description of the compiler.

Warning when Loading Compiled Files

CAUTION: Files compiled in Medley cannot be loaded back into Lyric. Medley-compiled .LCOM and .DFASL files will produce an error message when loaded into Lyric. (Lyric-compiled .LCOM and .DFASL files can be loaded and run in Medley.) If you need to run a Medley file in Lyric, load the source file and use the Lyric compiler.

Warning with Declarations

CAUTION: There is a feature of the BYTECOMPILER that is not supported by either the XCL compiler or SEdit. 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. The XCL compiler does not directly support this feature. If the body of the function gets DWIMIFIED for some other reason, the record declarations will happen to be noticed, otherwise they will not be seen and the compiler will signal an error if it can’t find an appropriate top-level record definition.

There are two caveats that you should note:

1. The compiler will give error messages "undefined record name ..." for the records that are declared this way, but will generate correct code.

2. SEdit does not recognize such declarations. Thus, if the "Expand" command is used in SEdit, the expansion will not be done with these record declarations in effect. The code that you see in the editor will not be the same code compiled by the BYTECOMPILER.

Section 18.3 Local Variables and Special Variables

(II:18.5)

The new execs always use the Common Lisp interpreter, causing LET and PROG statements at top level, particularly in a so-called Interlisp exec, to create lexical bindings, rather than deep or "special" bindings This can be worked around by setting **il:specvars** to T, which will cause the interpreter to create special bindings for all variables. This can also be worked around by wrapping the form to be "interlisp evaluated" in the IL:INTERLISP special form, which causes the Interlisp interpreter to be invoked.

Chapter 19 Masterscope

Masterscope is now a Lisp Library Module, not part of the environment.

Chapter 21 CLISP

CLISP infix forms do not work under the Common Lisp evaluator; only "clean" CLISP prefix forms are supported. You should run DWIMIFY in Koto on all other CLISP code before attempting to load it in Lyric. The remainder of this note describes the specific limitations on CLISP in Lyric.

There are two broad classes of transformations that DWIM applies to Lisp code:

1. A sort of macro expander that transforms **IF**, **FOR**, **FETCH**, etc. forms into "pure" Lisp code in well-defined ways.

2. A heuristic "corrector" that performs spelling correction and transforms CLISP infix forms such as X+Y into (PLUS X Y), sometimes having to make guesses as to whether X+Y might really have been the name of a variable.

An operational way of distinguishing the two is that DWIMIFY applied to code of type (1) makes no alterations in the code, whereas for code of type (2) it physically changes the form. Another difference is that code of type (2) must be dwimified before it can be compiled (user typically sets **DWIMIFYCOMPFLG** to **T**), whereas the compiler is able to treat code of type (1) as a special kind of macro.

Broadly speaking, code of type (2) is no longer fully supported. In particular, DWIM is invoked only when the code is encountered by the Interlisp evaluator. This means code typed to an "Old Interlisp" Executive, and code inside of an interpreted Interlisp function. Furthermore, some CLISP infix forms no longer DWIMIFY correctly. It is likely that CLISP infix will not be supported at all in future releases.

Expressions typed to the new Executives and inside of Common Lisp functions are run by the Common Lisp evaluator (**CL:EVAL**). As far as this evaluator is concerned, DWIM does not exist, and forms beginning with "CLISP" words **(IF**, **FOR**, **FETCH,** etc) are macros. These macros perform no DWIM corrections, so all of the subforms must be correct to begin with. This is a change from past releases, where the DWIM expansion of a CLISP word form also had the side effect of transforming any CLISP infix that it might have contained. For example, the macro expansion of

```
(if X then Y+1)
```

treats Y+1 as a variable, rather than as an addition. The correct form is

```
(if X then (PLUS Y 1)),
```

which is the way an explicit call to DWIMIFY would transform it.

If you have CLISP code from Koto you are advised to DWIMIFY the code before attempting to run or compile it in Lyric. Because of differences in the environments, not all CLISP constructs will DWIMIFY correctly in Lyric. In particular, the following do not work reliably, or at all:

1. The list-composing constructs using `<` and `>` do not DWIMIFY if the `<` is unpacked (an isolated symbol), because in Common Lisp, `<` is a perfectly valid CAR of form. On the other hand, the closing `>` *must* be unpacked if the last list element is quoted, since, for example, `(<A 'B>)` reads as `(<A (QUOTE B>)`).

2. Because of the conventional use of the characters `*` and `-` in Common Lisp names, those characters are only recognized as CLISP operators when they appear unpacked.

3. On the other hand, the operators + and / are the names of special variables in Common Lisp (Steele, p. 325), and hence cause no error when passed unpacked to the evaluator. Thus `(LIST X + Y)` returns a list of three elements, with no resort to DWIM; however, the parenthesized version `(LIST (X + Y))` and the packed version `(LIST X+Y)` both work.

If you routinely DWIMIFY code, so that no CLISP infix forms (type 2 above) remain on your source files, you may not need to make any changes. However, note that the fact that DWIMIFY of prefix forms implicitly performed infix transformations can hide code that escaped being completely dwimified before being written to a file.

There is a further caution regarding even routinely dwimified code that has not been edited since before Koto. Two uses of the assignment operator (_) no longer work, if not explicitly dwimified, because their canonical form (the output of DWIMIFY) has changed, and the old form is no longer supported when the form is simply evaluated, macro-expanded, or compiled (with **DWIMIFYCOMPFLG** = **NIL**):

1. Iterative statement bindings must always be lists. For example, the old form

```
(bind X_2 for Y in --)
```

is now canonically

```
(bind (X _ 2) for Y in --).
```

2. In a WITH expression, assignments must be dwimified to remove _. For example, the old form

```
(with MYRECORD MYFIELD _ (FOO))
```

is now canonically

```
(with MYRECORD (SETQ MYFIELD (FOO))).
```

DWIMIFY in Koto correctly made these transformations; however, in some older releases, it did not. Such old code must be explicitly dwimified (which you can do for these cases in Lyric). The errors resulting from failure to do so can be subtle. In particular, the compiler issues no special warning when such code is compiled. For example, in case 1, the macro expansion of the old form treats the symbol `X_2` as a variable to bind, rather than as a binding of the variable `X` with initial value 2. The only hint from the compiler that anything is amiss is likely to be an indication that the variable `X` is used freely but not bound. Case 2 is even subtler: the symbols `MYFIELD` and _ are treated as symbols to be evaluated; since their values are not used, the compiler optimizes them away, reducing the entire expression to simply `(FOO)`, and there is thus no warning of any sort from the compiler.

Chapter 22 Performance Issues

Section 22.3 Performance Measuring

(II:22.8)

The Interlisp-D **TIME** function has been withdrawn and replaced with the Common Lisp **TIME** macro (the symbol **TIME** is shared between IL and CL and thus need not be typed with a package prefix). The functionality of the *TIMEN* and *TIMETYP* arguments to the old **TIME** can be had by keywords to the **TIME** macro. The *Common Lisp Implementation Notes* describe the new **TIME** macro and its associated command in more detail.

[This page intentionally left blank]

VOLUME III—INPUT/OUTPUT

Chapter 24 Streams and Files

The Xerox Common Lisp file system supports multiple streams open simultaneously on the same file. This is an *incompatible change* to the semantics of Interlisp-D. You may have to modify old programs if they have not followed the guidelines listed in Sec 24.5 of the *Interlisp-D Reference Manual*. Some of the implications of this change for Interlisp programs are described below.

In prior releases of Interlisp-D, the system treated the *name* of an open file as a synonym for the *stream* open on the file. This meant that only one stream could be open at any time on a given file. In the Lyric release, a file name is no longer a unique name for an open stream. Thus, file names are no longer acceptable as the file/stream argument to any input/output or file system function that operates on an open stream (**READ, PRINT, CLOSEF**, **COPYBYTES**, etc). The only non-stream values acceptable as stream designators are the symbols **NIL** and **T**, designating the primary and terminal input/output streams. An attempt to use a litatom, even a "full file name," as a stream designator signals the error "LITATOM 'streams' no longer supported." Strings no longer designate an input stream whose source is the string itself—programs should call **OPENSTRINGSTREAM** instead, or use the comparable Common Lisp forms, such as **CL:WITH-INPUT-FROM-STRING**.

The functions **OPENFILE** and **OPENSTREAM** are now synonymous—both return a stream instead of a "full file name." The functions **INPUT** and **OUTPUT** also return streams. One exception to this is that **INPUT** and **OUTPUT** return **T** in the case where the primary input or output stream was previously directed to the terminal. However, this special behavior is for the Lyric release only; we recommend that you convert old code that depended on testing **(EQ (OUTPUT) T)**. Note that the values of the variables ***STANDARD-INPUT*** and ***STANDARD-OUTPUT*** are always streams, even if they are directed to the terminal.

The function **FULLNAME** can be used to obtain the name of a stream. For your convenience, the print syntax of streams now includes the name of the stream (if to a file) and its access (input, output, etc.). Functions, such as **UNPACKFILENAME**, that manipulate file names generally accept a stream as well, extracting the name of the file from the stream.

INFILEP still returns a full file name, as it is merely recognizing a file, not opening a stream to it. Programmers should be wary of code that subsequently tries to use the value of **INFILEP** as a stream argument. And, of course, the *FILENAME* argument to **OPENSTREAM** is still a name (a symbol or string), not a stream. **OPENSTREAM** also accepts a Common Lisp pathname as its *FILENAME* argument.

The function **CLOSEALL** is no longer implemented. The function **OPENP** returns **NIL** when passed a file name (or anything else but an open stream). However, for the Lyric release, **(OPENP NIL)** still returns a list of all streams open to files.

The functions **GETFILEINFO** and **SETFILEINFO** can still be given either an open stream or a file name. However, in the latter case, the request refers to the file, not to any stream open on the file. Thus, requesting the value of the attribute **LENGTH** for a file name may return a different value than requesting the value of the attribute **LENGTH** for a stream currently open on the file. **GETFILEINFO** returns **NIL** if given a file name and an attribute that only makes sense for streams (e.g., **ACCESS**, **ENDOFSTREAMOP**).

There is no difference between Common Lisp and Interlisp streams. A stream opened by an Interlisp function can be passed as argument to a Common Lisp input/output function, and vice versa.

Even though multiple streams per file are supported, the streams must still obey consistent access rules. That is, if a stream is open for output, no other streams on that file can be opened. It is not possible to **RENAMEFILE** or **DELFILE** a file that has *any* open stream on it.

The RS-232 or TTY ports are inherently single-user devices (rather than real files) thus, multiple streams cannot be open simultaneously on RS-232 or TTY.

Section 24.15 Deleting, Copying, and Renaming Files

(III:24.15)

The support of multiple streams per file now makes it possible to use **COPYFILE** without worrying about there being other readers of the file, in particular even when there is already a stream open on the file for sequential-only access (a case that failed in prior releases). Of course, **COPYFILE** cannot be used if the file already has an *output* stream open.

Chapter 25 Input/Output Functions

Variables Affecting Input/Output

There are several implicit parameters that affect the behavior of the input/output functions: the numeric print base, the primary output file, etc. In Common Lisp, these parameters are controlled by binding special variables. In Interlisp they are controlled by a functional interface; e.g., an output expression is wrapped in `(RESETFORM (RADIX 8) --)` to cause numbers to be printed in octal.

Where the input/output parameters in Common Lisp and Interlisp have essentially the same semantics, they have been integrated in

Lisp. That is, binding the Common Lisp special variable and calling the Interlisp function are equivalent operations, and they affect both Interlisp and Common Lisp input/output. However, it is considerably more efficient to bind a special variable than to call a function in a **RESETFORM** expression. In addition, binding a variable has only a local effect, whereas calling a function to side-effect the input/output parameters can also affect other processes. Thus, you are encouraged to use special variable binding to change parameters formerly changed via functional interface.

All of these variables are accessible in both the Common Lisp and Interlisp packages, so no package qualifier is required when typing them.

These parameters are as follows:

PRINT-BASE* vs RADIX** Binding ***PRINT-BASE to an integer *n* from 2 to 36 tells the printing functions to print numbers in base n. This is equivalent to (**RADIX** *n*). Note: this variable should not be confused with ***PRINT-RADIX***, another Common Lisp variable that controls whether Common Lisp functions include radix specifiers when printing numbers.

STANDARD-INPUT* vs INPUT** Binding ***STANDARD-INPUT to an input stream specifies the stream from which to read when an input function's stream argument is **NIL** or omitted. Evaluating ***STANDARD-INPUT*** is the same as evaluating (**INPUT**), except that (**INPUT**) returns **T** if the primary input for the process is the same as the terminal input stream (this compatibility feature is for the Lyric release only).

STANDARD-OUTPUT* vs OUTPUT** Binding ***STANDARD-OUTPUT to an output stream specifies the stream to which to print when an output function's stream argument is NIL or omitted. Evaluating ***STANDARD-OUTPUT*** is the same as evaluating (**OUTPUT**) except that (**OUTPUT**) returns **T** if the primary output for the process is the same as the terminal output stream (this compatibility feature is for the Lyric release only).

PRINT-LEVEL* & *PRINT-LENGTH* vs PRINTLEVEL** Binding ***PRINT-LEVEL to a positive integer *a* and ***PRINT-LENGTH*** to a positive integer *d* is equivalent to calling (**PRINTLEVEL** *a d*). Binding either variable to **NIL** is equivalent to specifying a value of -1 for the corresponding argument to **PRINTLEVEL,** i.e., it specifies infinite depth or length. Note that in Interlisp, print level is "triangular"—the print length decreases as the depth increases. In Common Lisp, the two are independent. Thus, although print level for both Interlisp and Common Lisp is controlled by a common pair of variables, the Interlisp and Common Lisp print functions interpret them (specifically ***PRINT-LENGTH***) slightly differently. In addition, Interlisp observes print level only when printing to the terminal, whereas Common Lisp observes it on all output.

READTABLE* vs SETREADTABLE** Binding ***READTABLE to a readtable specifies the readtable to use in any input/output function with a readtable argument omitted or specified as **NIL**. Evaluating ***READTABLE*** is the same as evaluating (**GETREADTABLE**). There is no longer a "NIL" or "T" readtable in Interlisp. See the discussion of readtables for more details.

Although the binding style is to be preferred to the **RESETFORM** expression, one difference should be noted with respect to error checking. In a form such as

(**RESETFORM** (RADIX *n*)
some-printing-code)

the value of *n* is checked immediately for validity, and an error is signalled if *n* is not an integer between 2 and 36. However, in

(**LET** ((***PRINT-BASE*** *n*))
some-printing-code)

there is no error checking at the time of the binding; rather, an error will not be signalled until the code attempts to print a number.

Similarly, the values of ***STANDARD-INPUT*** and ***STANDARD-OUTPUT*** must be actual streams, not the values that print functions coerce to streams, such as **NIL**, **T** or window objects.

Integration of Common Lisp and Interlisp Input/output Functions

Common Lisp and Interlisp have slightly different rules for reading and printing, regarding such things as escape characters, case sensitivity and number format. Each has two kinds of printing function, an escaped version (intended for reading back in) and an unescaped version. In order that Common Lisp and Interlisp programs can more freely intermix, Xerox Lisp isolates most of the reading/printing differences in the readtables used by both languages, rather than in the functions themselves. The exact rules have been chosen as a reasonable compromise between backward compatibility with Interlisp and integration with Common Lisp. This section outlines the details of this integration.

In what follows, the phrase "the readtable" generally refers to the readtable in force for the read or print operation being discussed. Specifically, this means an explicit readtable (other than NIL or T) passed as readtable argument to an Interlisp function, or else the current binding of ***READTABLE***. See the section on readtables for more details.

Section 25.2 Input Functions

The functions **IL:READ** and **CL:READ**, given the same readtable, interpret an input in exactly the same way. That is, the functions obey Common Lisp syntax rules when given a Common Lisp readtable, and Interlisp syntax when given an Interlisp readtable. Thus, the principal difference between the two is in the functionality provided by **CL:READ**'s extra arguments: end of file handling and the ability to specify that the read is recursive, which is mostly important when reading input containing circular structure references (the ## and #= macros). See *Common Lisp, the Language* for details of **CL:READ**'s optional arguments.

There is one further difference between **IL:READ** and **CL:READ**, in the handling of the terminating character. If the read terminates on a white space character, **CL:READ** consumes the character, while **IL:READ** leaves the character in the buffer, to be read by the next input operation. Thus, **IL:READ** is equivalent to **CL:READ-PRESERVING-WHITESPACE**. This difference is so that Interlisp

code that calls (**READC**) following a (**READ**) of a symbol will behave consistently between Koto and Lyric.

The Interlisp function **SKREAD** now defaults its readtable argument to the current readtable, viz., the value of ***READTABLE***, rather than **FILERDTBL**. This makes it consistent with all the other input functions, and is usually the correct thing, especially with the new reader environments used by the File Manager, but it is an incompatible change from Koto. **SKREAD** is also now implemented using Common Lisp's ***READ-SUPPRESS*** mechanism, which means that, unlike in Koto, it does something reasonable when it encounters read macros.

In the Medley release, reading in bitmaps from files is significantly faster.

Section 25.3 Output Functions

The discussion here is limited to the four basic printing functions: the unescaped and escaped Interlisp printing functions (**IL:PRIN1**, **IL:PRIN2**) and the corresponding Common Lisp functions (**CL:PRINC**, **CL:PRIN1**). All other print functions ultimately reduce to these. For example, **IL:PRINT** calls **IL:PRIN2**; **CL:FORMAT** with the ~S directive performs a **CL:PRIN1**.

IL:PRIN1 is essentially unchanged from previous releases. It uses no readtable at all, so is unaffected by the value of ***READTABLE***. It can be thought of as implicitly using the INTERLISP readtable.

Roughly speaking, **IL:PRIN2** and **CL:PRIN1** behave the same when given the same readtable. In particular, they both produce output acceptable to either **READ** function given the same readtable. Their minor differences are listed below.

CL:PRINC behaves like **CL:PRIN1**, except that it never prints escape characters or package prefixes. Thus, unlike **IL:PRIN1**, it *is* affected by the value of ***READTABLE***.

For the benefit of user-defined print functions, **IL:PRIN2** and **CL:PRIN1** bind ***PRINT-ESCAPE*** to **T,** while **IL:PRIN1** and **CL:PRINC** bind it to **NIL**. Thus, the print function can always examine ***PRINT-ESCAPE*** to decide whether it needs to print objects in a way that will read back correctly (Common Lisp user print functions may want to use **CL:WRITE** to pass ***PRINT-ESCAPE*** through transparently; Interlisp functions should choose **IL:PRIN2** or **IL:PRIN1** appropriately).

Printing Differences Between IL:PRIN2 and CL:PRIN1

There are two respects in which the Interlisp print functions (both **IL:PRIN1** and **IL:PRIN2**) differ from the Common Lisp ones, independent of readtable:

Line Length. The Interlisp functions respect the output stream's line length, while the Common Lisp functions all ignore it (they never insert newline characters when output approaches the right margin).

Print Level. The Interlisp functions respect the print level variables only when printing to the terminal (unless **PLVLFILEFLG** is true, see the *Interlisp-D Reference Manual*) or when printing with a Common Lisp readtable, whereas the Common Lisp functions respect them on *all* output.

Internal Printing Functions

Interlisp has several functions (e.g., **NCHARS**, **STRINGWIDTH**, **CHCON**, **MKSTRING**) that operate on the "prin1 pname" of an object, or optionally on its "prin2 pname" when given an extra flag and readtable as arguments. These functions are essentially unchanged in Lyric.

In terms of the discussion above, the "prin1 pname" of an object continues to be the characters that would be produced by a call to **IL:PRIN1** at infinite print level and line length, and with ***PRINT-BASE*** bound to 10 (unless **PRXFLG** is true, see *Interlisp-d Reference Manual*). The "prin2 pname" of an object is the list of characters that would be produced by a call to **IL:PRIN2** (or **CL:PRIN1**) using the specified readtable (or ***READTABLE*** if none is given), again at infinite print level and line length.

Exception: the function **STRINGWIDTH** computes the width of the expression as if it were printed at the current ***PRINT-LEVEL*** and ***PRINT-LENGTH***.

Printing Differences between Koto and Lyric

The Common Lisp and Interlisp printing functions use the same strategy for escaping characters in symbol names. Because of this, symbols may print differently in Lyric than they did in Koto, for two reasons: the use of the Common Lisp multiple escape character, and the escaping of numeric print names. Although the appearance is different, the functionality is the same—symbols are still printed in a way that allows them to be correctly read.

Roughly speaking, the multiple escape character is used to escape symbol names that would require two or more single escape characters. Thus, for example, a symbol that printed as `%(OH% NO%)` in Koto will print in Lyric as `|(OH NO)|`. However, in the old readtables that lack a multiple escape character (e.g., OLD-INTERLISP-T), the single escapes are still used. Multiple escapes are also used to print a symbol containing lower-case letters when the readtable is case-insensitive, e.g., `|Small|` in a Common Lisp readtable. Keep in mind also that some additional characters are now "special", e.g., colon in all new readtables, semi-colon in Common Lisp. Thus, the typical NS FIle "full name" will be printed with the multiple escape character.

Since it is now possible to create symbols that have "numeric" print names, such symbols must be printed with suitable escape characters, so that on input they are not misinterpreted as numbers. For example, the symbol whose print name is "1.2E3" is printed as `|1.2E3|`. In read tables lacking a multiple escape character, the single escape character is used instead, e.g., `%1.2E3` in the old Interlisp T readtable. A print name is considered

numeric according to the definition of "potential number" in Common Lisp (p. 341). Even if such a symbol is not readable in the current system as a number, it still needs to be escaped in case it is read into another system that treats it as numeric (either another Common Lisp implementation, or a future implementation of Xerox Lisp). Thus, some old Interlisp symbols now print escaped where they didn't in Koto; e.g., the **PRINTOUT** directive `|.P2|` is a potential number.

Bitmap Syntax

Bitmaps are printed in a new syntax in Lyric. When ***PRINT-ARRAY*** is **NIL** (the default at top level), a bitmap prints in roughly the same compact form as previously:

`#<BITMAP @` *nn*,*nnnnnn*`>`

If ***PRINT-ARRAY*** is **T**, a bitmap prints in a manner that allows it to be read back:

`#*(`*Width Height* [*BitsPerPixel*]`)` *XXXXXXXXX*...

Width and *Height* are measured in pixels; *BitsPerPixel* is supplied for bitmaps of other than the default of 1 bit per pixel. Each *X* represents four bits of a row of the bitmap; the characters `@` and `A` through `O` are used in this encoding. Thus, there are 4*⌈*Width***BitsPerPixel*/16⌉ *X*'s for each row.

MAKEFILE binds ***PRINT-ARRAY*** to **T** so that bitmaps print readably in files. E.g., if the value of FOO is a bitmap, the command (VARS FOO) dumps something like

```
(RPAQQ FOO #*(10 10)ADSDKJFDKJH...)
```

Note that with this new format, bitmaps are readable even inside a complex list structure. This means you need no longer use the **UGLYVARS** command when dumping a list containing bitmaps if the bitmaps were previously the only "unprintable" part of the list.

Section 25.8 Readtables

(III:25.34)

The input/output syntaxes of Common Lisp and Interlisp differ in a few significant ways. For example, Common Lisp uses "\" as the escape character, whereas Interlisp uses "%". Common Lisp input is case-insensitive (lower-case letters are read as upper-case), whereas Interlisp is case-sensitive. In Xerox Lisp, these differences are accommodated by having different readtables for the two dialects. Which syntax is used for input or output depends on which readtable is being used (either as an explicit argument to the read/print function or by being the "current" readtable).

Interlisp readtables have been extended to include features of Common Lisp syntax. There is a registry of named readtables to make it easier to choose a readtable. The default Interlisp readtable has been modified to make it look a little closer to Common Lisp.

Also, Lisp has a new mechanism for maintaining read/print consistency. This means that even though Koto files may contain

characters that are now "special", such as colon, you need make no changes to them—the File Manager knows how to load them correctly. See *IRM*, Chapter 17, Reader Environments and File Manager for details of this mechanism.

Differences Between Interlisp and Common Lisp Read Tables

When reading or printing, the readtable dictates the syntax rules being followed. As in past releases, the readtable indicates which characters must be escaped when printing a symbol (and ***PRINT-ESCAPE*** is true). In addition, in Lyric the readtable specifies such things as which escape character to use (\ or %) and the package delimiter to print on package-qualified symbols. The less obvious rules are detailed below.

Printing numbers. Numbers are always printed in the current print base (the value of the variable ***PRINT-BASE***, or equivalently the value of (**RADIX**). Whether to print a radix specifier is determined by the readtable. In Common Lisp, a radix specifier is printed exactly when the value of ***PRINT-RADIX*** is true. The radix specifier is a suffix decimal point in base 10, or a prefix using # for other bases. In Interlisp, a radix specifier is printed if the base is not 10, ***PRINT-ESCAPE*** is true, and the number is not less than the print base. The radix specifier is a suffix Q for octal, or a prefix using # (or | in old Interlisp readtables) for other bases. There is no decimal radix specifier.

Reading numbers. In Common Lisp, numbers are read in the current value of ***READ-BASE***, and a trailing decimal point is interpreted as a decimal radix specifier. In Interlisp, numbers are always read in base 10, and trailing decimal point denotes a floating-point number.

Case conversion. In a case-insensitive readtable (as Common Lisp is), the value of ***PRINT-CASE*** controls how upper-case symbols are printed, and lower-case letters in symbols are escaped. In a case-sensitive readtable (as Interlisp is), ***PRINT-CASE*** is ignored, so all letters in symbols are printed verbatim. ***PRINT-CASE*** is also ignored by **PRIN1**, which implicitly uses an Interlisp readtable.

Ratios. The character slash (/) is interpreted as the ratio marker in all readtables except old Interlisp readtables (specifically, those whose **COMMONNUMSYNTAX** property is **NIL**). This is so that old files containing symbols with slashes are not misinterpreted as ratios. Thus, the characters "1/2" are read in new readtables as the ratio 1/2, but in old Interlisp readtables as the 3-character symbol |1/2| (| is the multiple-escape character, see below). Ratios are printed in old Interlisp readtables in the form |.(/ numerator denominator).

Packages. Symbols are interned with respect to the current package (the binding of ***PACKAGE***) except in old Interlisp readtables (specifically, those whose **USESILPACKAGE** property is **T**), where symbols are read with respect to the INTERLISP package, independent of the binding of ***PACKAGE***. Again, this is a backward-compatibility feature: Interlisp had no package system, so programmers were not confronted with the need to read and print in a consistent package environment.

Print Level elision. When ***PRINT-LEVEL*** or ***PRINT-LENGTH*** is exceeded, the printing functions denote elided elements and elided tails by printing "&" and "--" with an Interlisp readtable, or "#" and "..." with a Common Lisp readtable.

Section 25.8.2 New Readtable Syntax Classes

The following new syntax classes are recognized by **GETSYNTAX** and **SETSYNTAX**:

MULTIPLE-ESCAPE This character inhibits any special interpretation of all characters (except the single escape character) up until the next occurrence of the multiple escape character. In Common Lisp and in the new Interlisp readtables this character is the vertical bar ("|"). For example, |(a)| is read as the 3-character symbol "(a)"; |x\|y\\z| is read as the 5 character symbol "x|y\z".

There is no multiple escape character in the old Interlisp readtables.

PACKAGEDELIM This character separates a package name from the symbol name in a package-qualified symbol. In Common Lisp and in the new Interlisp readtables this character is colon (":"). In the old Interlisp readtables the package delimiter is control-↑ ("↑↑"); it is not intended to be easily typed, but exists only to have a compatible way to print package-qualified symbols in obsolete readtables. See *Common Lisp, the Language* for details of package specification.

Additional Readtable Properties

Read tables have several additional properties in Xerox Lisp. These are accessible via the function **READTABLEPROP**:

(READTABLEPROP *RDTBL PROP NEWVALUE*) [Function]

Returns the current value of the property *PROP* of the readtable *RDTBL*. In addition, if *NEWVALUE* is specified, the property's value is set to *NEWVALUE*. The following properties are recognized:

NAME The name of the readtable (a string, case is ignored). The name is used for identification when printing the readtable object itself, and can be given to the function **FIND-READTABLE** to retrieve a particular readtable.

CASEINSENSITIVE If true, then unescaped lower-case letters in symbols are read as upper-case when this readtable is in effect. This property is true by default in Common Lisp readtables and false in Interlisp readtables.

COMMONLISP If true, then input/output obeys certain Common Lisp rules; otherwise it obeys Interlisp rules. This is described in more detail in the section on reading and printing. Setting this property to true also sets **COMMONNUMSYNTAX** true and **USESILPACKAGE** false.

COMMONNUMSYNTAX If true, then the Common Lisp rules for number parsing are followed; otherwise the old Interlisp rules are used. This affects the interpretation of "/" and the floating-point exponent specifiers "d", "f", "l" and "s". It does not affect the interpretation of decimal point and ***READ-BASE***, which are controlled by the **COMMONLISP** property. **COMMONNUMSYNTAX** is true for Common Lisp

readtables and the new Interlisp readtables; it is false for old Interlisp readtables.

USESILPACKAGE This is a backward compatibility feature. If **USESILPACKAGE** is true, then the Interlisp input/output functions read and print symbols with respect to the Interlisp package, independent of the current value of ***PACKAGE***. This property is true by default for old Interlisp readtables and false for others.

The following properties let the print functions know what characters are being used for certain variable syntax classes so that they can print objects in a way that will read back correctly. Note that it is possible for several characters to have the same syntax on input, but only one of the characters is used for output. Also note that only the three syntax classes **ESCAPE**, **MULTIPLE-ESCAPE** and **PACKAGEDELIM** are parameterized for output; the others (such as **LEFTPAREN** and **STRINGDELIM**) are hardwired —the same character is always used.

ESCAPECHAR This is the character code for the character to use for single escape. Setting this property also gives the designated character the syntax **ESCAPE** in the readtable.

MULTIPLE-ESCAPECHAR This is the character code for the character to use for multiple escape. Setting this property also gives the designated character the syntax **MULTIPLE-ESCAPE** in the readtable.

PACKAGECHAR This is the character code for the package delimiter. Setting this property also gives the designated character the syntax **PACKAGEDELIM** in the readtable.

(**FIND-READTABLE** *NAME*) [Function]

Returns the readtable whose name is *NAME*, which should be a symbol or string (case is ignored); returns **NIL** if no such readtable is registered. Readtables are registered by calling (**READTABLEPROP** *rdtbl* 'NAME *name*).

(**COPYREADTABLE** *RDTBL*) [Function]

COPYREADTABLE has been extended to accept a readtable name as its *RDTBL* argument (the old value ORIG could be considered a special case of this). For example, (**COPYREADTABLE** "INTERLISP") returns a copy of the INTERLISP readtable. **COPYREADTABLE** preserves all syntax settings and properties except NAME.

Section 25.8 Predefined Readtables

The following readtables are registered in the Lyric release of Lisp:

INTERLISP This is the "new" Interlisp readtable. It is used by default in the Interlisp Exec and by the File Manager to write new versions of pre-existing source files. It thus replaces the old T readtable, FILERDTBL, CODERDTBL and DEDITRDTBL. It differs from them in the following ways:

| (vertical bar) has syntax **MULTIPLE-ESCAPE** rather than being used as a variant of the Common Lisp dispatching # macro character.

is used as the Common Lisp dispatching # macro character. For example, to type a number in hexadecimal, the syntax is #xnnn rather than |xnnn.

: (colon) has syntax **PACKAGEDELIM**.

' (quote) reads the next expression as (QUOTE expression).

` (backquote)
, (comma) are used to read backquoted expressions

In addition, the Common Lisp syntax for numbers is supported (the readtable has property **COMMUNNUMSYNTAX**). For example, the characters "1/2" denote a ratio, not a symbol. Note, however, that trailing decimal point still means floating point, rather than forcing a decimal read base for an integer.

The syntax for quote, backquote, and comma is the same as in OLD-INTERLISP-T, so you will not see any difference when typing to an Interlisp Exec. However, the fact that files are also written in the new INTERLISP readtable means that the prettyprinter is now able to print quoted and backquoted expressions much more attractively on files (and to the display as well).

LISP This readtable implements Common Lisp read syntax, exactly as described in *Common Lisp, the Language*.

XCL This readtable is the same as LISP, except that the characters with ASCII codes 1 thru 26 have white-space (**SEPRCHAR**) syntax. This readtable is intended for use in File Manager files, so that font information can be encoded on the file.

The following readtables are provided for backward compatibility. They are the same as the corresponding readtables in the Koto release, with the addition of the **USESILPACKAGE** property.

ORIG This is the same as the ORIG readtable described in the *Interlisp-D Reference Manual*. When using a readtable produced by (**COPYREADTABLE** 'ORIG), expressions will read and print exactly the same in Koto and Lyric.

OLD-INTERLISP-FILE This is the same as the FILERDTBL described in the *Interlisp-D Reference Manual*. This readtable is used to read source files produced in the Koto release. Note that in Lyric, FILERDTBL is no longer used when reading or writing new files; see the section on reader environments.

OLD-INTERLISP-T This is the same as the T readtable described in the *Interlisp-D Reference Manual*.

If you wish to change the syntax used by a standard readtable, it is recommended instead that you copy the readtable, give it a distinguished name, and make the change in the new readtable. This will reduce the likelihood that you will try to read another user's files in an incompatible readtable, or that another user will fail reading yours. See chpater 17, Reader Environments and the File Manager, for more details.

Koto Compatibility Considerations

In order to consistently read a data structure that you have previously printed, it is important that **READ** and **PRINT** both use the same readtable and package. Code that calls **READ** or **PRINT** without explicitly specifying a readtable (via the *RDTBL* argument or by doing a **SETREADTABLE**) is thus in some danger of reading and printing inconsistently.

Specifying Readtables and Packages

In Koto, the "primary" (NIL) readtable was not significantly different from the other Interlisp readtables, and users tended not to make significant modifications to the primary readtable anyway. As a result, it was easy to write code that was not careful about readtables and get away with it. In Lyric, however, there are significant differences among commonly used readtables. Thus, if code using the default readtable called **PRINT** under, say, the Common Lisp Executive and tried to **READ** the expression back while running under an Interlisp Executive, it might very well get inconsistent results.

Lyric also introduces the extra complication of the default package, which is the other important parameter affecting the behavior of **READ** and **PRINT**.

Programmers are thus advised to fix any code that uses **READ** and **PRINT** as a way of storing and retrieving Lisp expressions to be sure to specify a readtable and package environment. For new code in Lyric, this can be done by binding the special variables ***READTABLE*** and ***PACKAGE***. If it is necessary to write code that works in both Koto and Lyric, the programmer should pass an explicit readtable to all **READ** and **PRINT** functions, or set the primary readtable using (**RESETFORM** (**SETREADTABLE** *rdtbl*) --). If the readtable chosen is either FILERDTBL or one derived as a copy of ORIG, then **READ** and **PRINT** will automatically use the INTERLISP package in Lyric, thereby avoiding any need to specify a binding for ***PACKAGE***.

The T Readtable

An additional possible incompatibility exists with regard to the Koto T readtable: The T readtable was "the readtable used when reading from the terminal". In Lyric, the T readtable is synonymous with NIL, and all Executives bind ***READTABLE*** to the appropriate value for the Exec. This is unlikely to be a major source of incompatibility, as few programs depend on printing something in the T readtable in a way that needs to read back consistently.

PQUOTE Printed Files

In Lyric, the prettyprinter automatically prints quoted and backquoted expressions attractively. Hence, the PQUOTE Lispusers module is now obsolete. However, if you have written files in the past with the PQUOTE module loaded into your environment, you need to do the following in Lyric in order to load those files:

(**SETSYNTAX (CHARCODE "'") '(MACRO FIRST READQUOTE) FILERDTBL)**

You can then load the old files. New files produced in Lyric by **MAKEFILE** will automatically be loadable, so you need only perform the **SETSYNTAX** change as long as you still have old files written with PQUOTE. Remember, of course, that as long as the **SETSYNTAX** is in effect (as with the old PQUOTE module), if you read old files that were written without PQUOTE you may read them incorrectly.

Back-Quote Facility

The back-quote facility now fully conforms with *Common Lisp the Language*. This means some cases of nested back-quote now work correctly. Back-quote forms are also more attractively displayed by the prettyprinter. Users should beware, however, that the back-quote facility does not attempt to create fresh list structures unless it is necessary to do so. Thus for example,

```
'(1 2 3)
```

is equivalent to

```
'(1 2 3)
```

not

```
(LIST 1 2 3)
```

If you need to avoid sharing structure you should explicitly use **LIST**, or **COPY** the output of the back-quote form.

Chapter 28 Windows and Menus

Section 28.5.1 Menu Fields

(III:28.38)

With the Medley release, multi-column menus can have rollout submenus.

[This page intentionally left blank]

4. CHANGES TO INTERLISP-D IN LYRIC/MEDLEY

NOTE: Chapter 4 is organized to correspond to the original *Interlisp-D Reference Manual*, and explains changes that have occurred in Interlisp-D with the Lyric and Medley releases. To make it easy to use this chapter with the *IRM,* information is organized by *IRM* volume and section numbers. Section headings from the *IRM* are maintained to aid in cross-referencing.

Lyric information as well as Medley release enhancements are included. Medley additions are indicated with revision bars in the right margin.

VOLUME I—LANGUAGE

Chapter 3 Lists

Section 3.2 Building Lists From Left To Right

(I:3.7)

The functions **DOCOLLECT** and **ENDCOLLECT** are no longer supported.

(I:3.8)

The description of the **ADDTOSCRATCHLIST** function has been revised to read:

(ADDTOSCRATCHLIST *VALUE)* [Function]

For use inside a **SCRATCHLIST** form. *VALUE* is added on to the end of the value being collected by **SCRATCHLIST.** When the **SCRATCHLIST** returns, its value is a list containing all of the things that **ADDTOSCRATCHLIST** has added.

Section 3.10 Sorting Lists

(I:3.17)

(**SORT** *DATE COMPAREFN*) [Function]

There is no safe interrupt to **SORT**—if you abort a call to **SORT** by *any* means the possibility exists for losing elements from the list being sorted.

Chapter 6 Hash Arrays

(I:6.1)

(**HASHARRAY** *MINKEYS OVERFLOW HASHBITSFN EQUIVFN RECLAIMABLE REHASH-THRESHOLD*) [Function]

The function **HASHARRAY** has two new optional arguments, *RECLAIMABLE* and *REHASH-THRESHOLD*. If *RECLAIMABLE* is true, then entries in the hash table are considered "reclaimable" in the sense that the system is permitted to remove any key and its associated value from the hash table at any time. In practice, the contract is less severe: the system only removes keys when a hash table fills and is about to be rehashed, and then it only removes keys whose reference count is one, and to which there are thus no pointers outstanding except possibly from the stack (local variables). This is useful for hash tables that serve to cache information about Lisp objects to avoid recomputation; for example, the system hash table CLISPARRAY is now reclaimable. Discarding keys keeps the table from necessarily needing to grow, and potentially allows the storage consumed by both the key and value to be reclaimed.

Section 6.1 Hash Overflow

(I:6.3)

You should note changes to the wording of two of the possibilites for the overflow method:

The first sentence for **NIL** should read: The array is automatically enlarged by *at least* a factor of 1.5 every time it overflows.

The explanation for "a positive integer N" should read: The array is enlarged to include *at least* N more slots than it currently has.

Chapter 7 Integer Arithmetic

(I:7.5)

The variables **MIN.INTEGER** and **MAX.INTEGER** have been removed from the *Interlisp-D Reference Manual*. Therefore, calling **(MIN)** and **(MAX)** is an error.

(***I:7.7***)

(**FIXR** *N*) [Function]

When N is exactly half way between two integers, FIXR rounds it to the even number. For example (FIXR 1.5) ⇒ 2 and (FIXR 2.5) ⇒ 2.

Section 7.3 Logical Arithmetic Functions

The function **INTEGERLENGTH** does *not* coerce floating point numbers to integers; rather, it signals an error, "Arg not Integer". (This was true in Koto as well.)

Section 7.5 Other Arithmetic Functions

(I:7.13)

The algorithms for **SIN**, **COS**, and other trigometric functions have been tuned and are now accurate to at least six significant figures.

Chapter 8 Record Package

(I:8.11)

When using **BLOCKRECORD**, it is an error to try to declare a record with a zero-length field. Previously, the system would go into an infinite loop. In the Medley release, the system will now detect this and signal an error.

Chapter 9 Conditionals and Iterative Statements

Section 9.2 Equality Predicates

(I:9.3)

(**EQUALALL** *X Y*) [Function]

Add the following NOTE to the **EQUALALL** function:

Note: In general, **EQUALALL** descends all the way into all datatypes, both those defined by the user and those built into the system. If you have a data structure with fonts and pointers to windows, **EQUALALL** will descend into those also. If the data structures are circular, as windows are, **EQUALALL** can cause a Stack Overflow error.

Section 9.8.3 Condition I.s. oprs

UNTIL *N* (*N* a number) [I.S. Operator]

REPEATUNTIL *N* (*N* a number) [I.S. Operator]

These descriptions were included in the *Interlisp-D Reference Manual* in error and should be removed. **UNTIL** and **REPEATUNTIL** work *only* with predicate expressions, not numbers.

Chapter 10 Function Definition, Manipulation , and Evaluation

Section 10.2 Defining Functions

(I:10.11)

In the definition of the **MOVD** function, the sentence "**COPYDEF** is a higher-level function that only moves expr definitions, but..." should be revised to read:

COPYDEF is a higher-level function that not only moves expr definitions, but also works for variables, records, etc.

Section 10.5 Functional Arguments

(I:10.19)

FUNARG functionality (non-**NIL** second argument to **FUNCTION**) has been withdrawn. Most of the uses for Interlisp **FUNARG**'s are better written using the lexical closure functionality of Common Lisp.

Section 10.6.2 Interpreting Macros

The variables **SHOULDCOMPILEMACROATOMS** and **UNSAFEMACROATOMS** no longer exist.

Chapter 11 Variable Bindings and the Interlisp Stack

(II:11.2)

In Lisp there is a fixed amount of space allocated for the stack. When this space is exhausted, the **STACK OVERFLOW** error occurs. However, if the system waited until the stack were *really* exhausted, there wouldn't be room to run the debugger. Thus, a portion of the stack space is reserved; when the stack intrudes into the reserved area, it causes a stack overflow interrupt, and subsequently a call to the debugger.

In order not to get a **STACK OVERFLOW** error while inside the debugger, this intrusion into the reserved area is only noted once. If the reserved area is exhausted, then a "hard" stack overflow occurs (a 9319 MP halt), from which the only recourse is a hard reset via STOP (or Ctrl-D from TeleRaid). Following a hard reset, the stack is cleared, stack overflow detection is reenabled, and all processes are restarted.

The implications of this are that you should not attempt any deep computations while inside the debugger for a stack overflow error, and you should call **(HARDRESET)** as soon as possible in order that subsequent stack overflows can again be caught in the debugger before they advance to the MP halt. In order to make this more convenient, the system automatically calls **(HARDRESET)** if you exit the debugger via the **^** or **OK** commands, or abort with a Ctrl-D. The only way to exit the debugger without having a

(HARDRESET) occur is by using the **RETURN** command. You can disable this feature by setting **AUTOHARDRESETFLG** to **NIL**, in which case you must be sure to perform the **(HARDRESET)** yourself if you want the next stack overflow to be detected early enough to enter the debugger.

Section 11.2.1 Searching the Stack

(**STKPOS** *FRAMENAME N POS OLDPOS*) [Function]

(STKPOS 'STKPOS) does not cause an error; it merely returns NIL. (This was true in Koto as well.) It is still not permissible to create a pointer to the active frame; however, **STKPOS** never attempts to, as it starts searching for the specified frame by looking first at its caller.

Section 11.2.2 Variable Bindings in Stack Frames

(I:11.7)

(**STKARG** *N POS* —) [Function]

(**STKNARGS** *POS* —) [Function]

The functions **STKARG** and **STKNARGS** will now return the number of arguments supplied to a Lambda Nospread when there is a break. The **?=** command will show all the arguments.

(**SETSTKARGNAME** *N POS NAME*) [Function]

The function **SETSTKARGNAME** does not work for interpreted frames.

Section 11.2.5 Releasing and Reusing Stack Pointers

(**CLEARSTK** *FLG*) [Function]

(CLEARSTK NIL) is a no-op—the ability to clear all stack pointers is inconsistent with the modularity implicit in a multi-processing environment.

CLEARSTKLST [Variable]

NOCLEARSTKLST [Variable]

The variables CLEARSTKLST and NOCLEARSTKLST are no longer used. (More precisely, they are used only by the Old Interlisp Executive, which means that programs can no longer depend on them.)

Section 11.2.7 Other Stack Functions

(II:11.13)

In the **REALFRAMEP** function, the *INTERPFLG* argument description has been corrected to read:

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**.

Chapter 12 Miscellaneous

Section 12.2 Idle Mode

The following properties in **IDLE.PROFILE** are new or have meanings different from the documentation in the *Interlisp-D Reference Manual*:

ALLOWED.LOGINS The authentication aspects of this property have been separated into the **AUTHENTICATE** property. The value of this property now speaks specifically to who is allowed to exit idle mode: If the value is **NIL** (or any other non-list), no login at all is required to exit Idle mode. Otherwise, the value is a list composed of any of the following:

***** Require login, but let anyone exit idle mode. This will overwrite the previous user's name and password each time idle mode is exited.

T Let the previous user (as determined by **USERNAME**) exit idle mode. If the user name has not been set, this is equivalent to ***.**

A user name Let this specific user exit idle mode.

A group name Allow any members of this group (an NS Clearinghouse group name) to exit idle mode.

The initial value for **ALLOWED.LOGINS** is **(T *)**, i.e., anyone is allowed to exit idle mode.

AUTHENTICATE The value of this property determines what mechanism is used to check passwords. If **T**, use the NS authentication protocol (requires the presence of an NS Authentication server accessible via the network). If **NIL**, do not check the password at all—accept any password. This is only particularly useful if **ALLOWED.LOGINS** contains ***.**

The initial value of **AUTHENTICATE** is **T**.

FORGET If this is the symbol **FIRST**, the user's password will be erased when idle mode is entered. Otherwise, this property is relevant only when **ALLOWED.LOGINS** is **NIL** (if **ALLOWED.LOGINS** is a list, then some sort of login is required, which will have the effect of erasing any previous login): If **FORGET** is non-**NIL**, the user's password will be erased when idle mode is exited. Initial value is **T** (erase password on exit).

Note: If the password is erased on *entry* to Idle mode (value **FIRST**), any program left running when idle mode is entered may fail if it tries doing anything requiring passwords (such as accessing file servers).

LOGIN.TIMEOUT Value is a number indicating how many seconds Idle's prompt for a login should remain up before timing it out and resuming Idle mode. Initial value is 30. This feature avoids the problem of having an Idle machine "freeze up" indefinitely (stop running the idle pattern) just because someone brushed against the keyboard.

RESETVARS This property is no longer used; rather, the value of the global variable **IDLE.RESETVARS** is used instead.

SUSPEND.PROCESS.NAMES This property is no longer used; rather, the value of the global variable **IDLE.SUSPEND.PROCESS.NAMES** is used instead.

Section 12.3 Saving Virtual Memory State

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. Its 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 occurring:

BEFORELOGOUT The system is about to perform a **LOGOUT**. The event function might want to save state, notify a network connection that it is about to go away, etc.

BEFORESYSOUT
BEFOREMAKESYS
BEFORESAVEVM The system is about to perform a **SYSOUT**, **MAKESYS**, or **SAVEVM**. Often these three events are treated equivalently; however, sometimes the distinction is interesting. For example, a program might want to perform a much more extensive tidying-up before **MAKESYS** than if it is merely doing a routine **SAVEVM**.

AFTERLOGOUT
AFTERSYSOUT
AFTERMAKESYS
AFTERSAVEVM The system is starting up a virtual memory image that was saved by performing a **LOGOUT**, **SYSOUT**, etc. Ordinarily, the event function should treat all of these the same—in all four cases, some arbitrary amount of time has passed, remote files may have come and gone, a different user may be logged in, or the virtual memory image might even be running on a different workstation.

AFTERDOSYSOUT
AFTERDOMAKESYS
AFTERDOSAVEVM The system is continuing in the same virtual memory image following a **SYSOUT**, **MAKESYS**, or **SAVEVM** (as opposed to having just booted the same virtual memory image). Ordinarily, these events are uninteresting; they exist solely so that actions taken by the **BEFORExxx** events can be compensated for after the event. For example, if the before event cleared a cache, the after event might initiate refilling it. There is, of course, no event **AFTERDOLOGOUT**, as **LOGOUT** does not "continue".

Section 12.4 System Version Information

(I:12.13)

In the description of the **MACHINETYPE** function, add another machine, the **DOVE** (for the Xerox 1186).

VOLUME II—ENVIRONMENT

Chapter 13 Interlisp Executive

(I:23.37)

(**READLINE** *RDTBL — —*) [Function]

The *Interlisp-D Reference Manual* states:

The description on p 13.37 of **READLINE**'s behavior when one or more spaces precede the carriage return applies only when **LISPXREADFN** is **READ**. **LISPXREADFN** is initially **TTYINREAD**, which ignores spaces before the carriage return, and thus will never prompt you with "..." for an additional line. Also, the new Executive does not use **READLINE** at all, so you will never see this behavior in a new Executive, independent of the setting of **LISPXREADFN**.

Chapter 14 Errors and Breaks

Section 14.5 Break Window Variables

(II:14.15)

Setting the variable **BREAKREGIONSPEC** to NIL no longer creates problems if there is a subsequent break.

Section 14.8 Catching Errors

(II:14.22)

The Nlambda functions **ERSETQ** and **NLSETQ** now allow evaluation of an arbitrary number of forms, rather than only one.

(II:14.26)

For Medley, the Interlisp interpreter's handler for **RESETFORM** has been fixed (in Lyric, it worked only from the Common Lisp interpreter, or compiled) .

Chapter 17 File Package

Note: The File Package is now known as the File Manager.

Section 17.8.1 Functions for Manipulating Typed Definitions

(II:17.26)

(**HASDEF** *NAME TYPE SOURCE SPELLFLG*) [Function]

Clarification: **HASDEF** for type FNS (or NIL) indicates that *NAME* has an editable source definition, not that *NAME* is defined at all. Thus if *NAME* exists on a file for which you have loaded only the compiled version but not the source, **HASDEF** returns **NIL**.

Section 17.8.2 Defining New File Package Types

(II:17.31)

In the **WHENCHANGED** File Package Type Property the *REASON* argument passed to **WHENCHANGED** no longer is **T** or **NIL**. The Note has been revised as follows:

Note: The *REASON* argument passed to **WHENCHANGED** functions is either **CHANGED** or **DEFINED**.

(II:17.32)

The Nospread Function **FILEPKGTYPE** returns a property list rather than an alist.

Section 17.9.2 Variables

(II:17.36)

In the Lyric release, **HORRIBLEVARS** did not preserve common substructures across several variables.

In Lyric, you could not dump an **UGLYVARS** or **HORRIBLEVARS** whose printed representation required more than *ARRAY-TOTAL-SIZE-LIMIT* characters. This is no longer the case with the Medley release.

Section 17.9.8 Defining New File Package Commands

(II:17.47)

The Nospread Function **FILEPKGCOM** returns a property list rather than an alist.

Section 17.11 Symbolic File Format

(**PRETTYDEF** *PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES*) [Function]

PRETTYDEF accepts only a symbol for its file argument.

In Lyric and Medley, SYSPRETTYFLG is ignored in Interlisp exec and does not pretty-print values in the executive.

(**LISPSOURCEFILEP** *FILE*) [Function]

LISPSOURCEFILEP is more specifically defined to mean that the file is in File Manager format *and* has a file map.

Section 17.11.3 File Maps

File maps are no longer stored on the FILEMAP property. See **GET-ENVIRONMENT-AND-FILEMAP** in the section entitled "Programmer's Interface to Reader Environments" in chapter 3.

Chapter 18 Compiler

CAUTION: Files compiled in Medley cannot be loaded back into Lyric. Medley-compiled .LCOM and .DFASL files will produce an error message when loaded into Lyric. (Lyric-compiled .LCOM and .DFASL files can be loaded and run in Medley.) If you need to run a Medley file in Lyric, load the source file and use the Lyric compiler.

Note that you should not attempt to compile a file containing a function named **STOP**. The format of the .LCOM file produced by **BCOMPL** or **TCOMPL** admits an unfortunate ambiguity in the treatment of the symbol **STOP**—**LOAD** prefers to treat it as the token signifying the end of the file, rather than as starting the definition of the function **STOP**.

There is no such restriction for the .DFASL files produced by **CL:COMPILE-FILE.**

Chapter 21 CLISP

Section 21.8 Miscellaneous Functions and Variables

(**CLEARCLISPARRAY** *NAME* — —) [*Function*]

Macro and CLISP expansions are cached in CLISPARRAY, the system's CLISP hash array. When anything changes that would invalidate an expansion, it needs to be removed from the cache. CLEARCLISPARRAY removes from the CLISP hash array any key whose CAR is *NAME*. The system does this automatically whenever you edit a clisp or macro form, or when you redefine a clisp word or macro definition. However, you may need to call CLEARCLISPARRAY explicitly if you change something in a more subtle way, e.g., you redefine a function used by a macro. If your change invalidates an unknown set of expansions, you might prefer to take the performance penalty of calling (CLRHASH CLISPARRAY) to invalidate the entire cache, just to make sure no incorrect expansions are kept around.

Chapter 22 Performance Issues

Section 22.1 Storage Allocation and Garbage Collection

The following should be appended to the description of garbage collection in Interlisp-D:

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.

Section 22.5 Using Data Types Instead of Records

(II:22.13)

The note in this section states that "pages for datatypes are allocated one page at a time." The note should read:

Space for datatypes is allocated two pages at a time. Thus, each datatype for which any instances at all have been allocated has at least two pages assigned to it.

Chapter 23 Processes

Section 23.1 Creating and Destroying Processes

(III:23.2)

ADD.PROCESS no longer coerces the process name to a symbol. Rather, process names are treated as case-insensitive strings. Thus, you can use strings for process names, and when typing process commands to an exec, you need not worry about getting the alphabetic case correct.

Section 23.2 Process Control Constructs

The Medley release fixes the **PROCESS.EVAL** and **PROCESS.APPLY** functions. In **PROCESS.EVAL** and **PROCESS.APPLY**, with argument *WAITFORRESULT* = **T**, if the computation in the other process aborts (or the process is killed), then **PROCESS.EVAL** and **PROCESS.APPLY** return **:ABORTED** instead of hanging.

Section 23.6 Typein and the TTY Process

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.

Section 23.8 Process Status Window

The Medley release modifies the way in which the Process Status Window can be reshaped and refreshed.

The Process Status Window is now created in such a way that reshaping the window reshapes ONLY the backtrace window, not the main window.

The process status window now refreshes itself automatically following a KILL command.

VOLUME III—INPUT/OUTPUT

Chapter 24 Streams and Files

Section 24.7 File Attributes

(**GETFILEINFO** *FILE ATTRIB*) [Function]

NS file servers implement the following additional attributes for **GETFILEINFO** (neither of these attributes is currently 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*), where *name* is the name of a user or group or a name pattern, and *rights* is one or more of the symbols ALL READ WRITE DELETE CREATE MODIFY. For servers running Services release 10.0 or later, *nametype* is the symbol "--"; in earlier releases it is either INDIVIDUAL or GROUP, to distinguish the type of name. For example, the value ((Jane Jones: -- ALL) (*: -- READ)) means that user Jane Jones has full access to the file, while all members of the default domain only have read access to the file.

Section 24.9 Local Hard Disk Device

(III:24.22)

In the Medley release, the {DSK} device now accepts a wider range of characters in file names. Almost any character in char set 0 is acceptable. Previously, if you tried to create a file whose name included, for example, an underscore, you would see a "FILE NOT FOUND" error.

Section 24.10 Floppy Disk Device

(III:24.26)

As of the Lyric release, CPM-format floppy disks are no longer supported.

Section 24.12 Temporary Files and CORE Device

(III:24.30)

In Medley, (**GETFILEINFO** xx 'LENGTH) works for both opened and closed **NODIRCORE** streams.

A closed **NODIRCORE** stream can be reopened.

Section 24.18.1 Pup File Server Protocols

UNIXFTPFLG [Variable]

When the Leaf protocol was first implemented for the Vax Unix operating system, its use was inconsistent with the operation of the Pup FTP server on the same host: the Leaf server supported versions, but the Ftp server knew only about the native, versionless file system. Thus, Lisp could not use the two protocols interchangeably. For example, if it used Ftp to write a file FOO, the Ftp server would, in versionless style, overwrite the versionless file FOO, rather than create a new version FOO;6 to supersede the highest version FOO;5 created by the Leaf server.

Lisp thus makes the conservative assumption that the Ftp server is unusable for anything other than directory enumeration on a host of type UNIX. This is unfortunate, since it is often the case that Ftp is more efficiently implemented than Leaf, since one need only tune the performance of sequential access.

More recent versions of the Unix Pup software have a Leaf and Ftp server more in agreement with each other. Setting **UNIXFTPFLG** to true (it is initially NIL) informs Lisp that all the Unix servers accessible on your internetwork that possess Ftp servers are safe to use in parallel with their Leaf servers.

Section 24.18.1 and 24.18.2 Use of BREAKCONNECTION with File Servers

(III:24.37)

In Medley, the function **BREAKCONNECTION** can be used equally well with NS servers and Leaf servers. Formerly, it only worked on Leaf servers, and there was a separate function (BREAK.NSFILING.CONNECTION *HOST*) to handle NS servers.

(BREAKCONNECTION *HOST FAST***)** [Function]

Breaks the file server connection to *HOST*. If *HOST* = T, breaks connections to all file servers that understand the **BREAKCONNECTION** method (currently Leaf and NS). **BREAKCONNECTION** returns the server name, or if *HOST* = T, returns a list of all hosts that responded to the **BREAKCONNECTION** request.

This function may be useful if Lisp and the server disagree about what files are open, or if the Lisp system is caching something that you do not want it to; e.g., if you get a file busy error from another workstation for a file that you may have touched on this workstation.

The behavior of **BREAKCONNECTION** is server-specific. On an NS server, **BREAKCONNECTION** releases any locks that Lisp may have on recently-accessed files, including those for open files, but does not close any files from Lisp's point of view--any subsequent access to an open file will quietly reestablish the connection. Most NS servers have a short timeout on the order of 10 minutes after which an implicit **BREAKCONNECTION** occurs if you have no files open.

On a Leaf server, **BREAKCONNECTION** first closes any files. If the argument *FAST* is true, it marks the files closed without attempting to close them cleanly. Leaf connections ordinarily do not timeout if any files at all are open.

Section 24.18.2 NS File Server Protocols

(III:24.37)

Medley incorporates the random access capability on NS servers provided by the NSRANDOM LispUsers module in Lyric.

The Medley release also supports NS file names containing characters other than character set 0 (e.g., Greek characters).

Section 24.18.3 Operating System Designations

DEFAULT.OSTYPE [Variable]

If a host's name is not found in **NETWORKOSTYPES**, its operating system type is assumed to be the value of **DEFAULT.OSTYPE**. This variable may be of use to sites with many servers all of the same type. Its default value (IFS) is, unfortunately, inappropriate for most sites. It is recommended you set **DEFAULT.OSTYPE** in the initialization file that lives on the local disk (*not* in an init file on a file server, since Lisp needs to know the operating system type before talking to the server).

Chapter 25 Input/Output Functions

Section 25.2 Input Functions

(LASTC *FILE***)** [Function]

The function **LASTC** can return an incorrect result when called immediately following a **PEEKC** on a file that contains run-coded NS characters.

Section 25.3.2 Printing Numbers

(III:25.15)

In the **PRINTNUM** function, the **FLOAT** format option **(FLOAT 7 2 NIL T)** is illegal; change the option to **(FLOAT 7 2 NIL 0)**.

Section 25.3.4 Printing Unusual Data Structures

(**HPRINT** *EXPR FILE UNCIRCULAR DATATYPESEEN*) [Function]

Using **HPRINT** to save structures that include pointers to raw storage will cause stack overflows. This includes dumping things using the **VARS**, **UGLYVARS**, or **HORRIBLEVARS** filemanager commands.

For example, a font descriptor points to raw storage, and cannot be dumped; for that reason, other system data types (e.g. windows) that point to fonts also cannot be dumped.

Section 25.4 Random Access File Operations

(III:25.20)

The first argument in the **FILEPOS** function should be called *STR* not *PATTERN*.

(III:25.20)

In the Medley release, the function **COPYBYTES** now accepts *START* and *END* arguments even when the input stream is not random access. This caused an error in earlier releases.

Section 25.6 PRINTOUT

(III:25.27)

The PRINTOUT command **.FONT** changes the **DSPFONT** font permanently, that is, even after printout finishes.

Section 25.8.3 READ Macros

(III:25.42-43)

These **READMACROS** appear only in the OLD-INTERLISP-T readtable. (See Section 2 for a description of Lyric readtables.)

Chapter 26 User Input/Output Packages

Section 26.3 ASKUSER

(**ASKUSER** *WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRNTFLG OPTIONSLST FILE*) [Function]

ASKUSER does not accept a string to mean a stream open on the string; you must call **OPENSTRINGSTREAM** if that's what you mean.

Section 26.4 TTYIN Display Typein Editor

(III:26.22)

The following fixes have been made to TTYIN in the Medley release:

- TTYIN now respects the **DSPLEFTMARGIN** of the ttydisplaystream, rather than assuming it is zero.
- You can now assign the keyaction 194 (octal 302--acute accent in the NS character set) to a key and TTYIN will not treat it like the UNDO key (except on the 1132, where this functionality is still on blank-middle).

- TTYIN correctly handles prompts that are wider than the window.
- TTYIN now handles NS characters correctly when you are using a fixed-width font into which you have coerced, say, Classic characters for the non-zero character sets.
- TTYIN now handles Escape completion much more efficiently. If the completion is ambiguous, it completes the unambiguous prefix (as it did in Koto but not Lyric); it also correctly interprets escape characters. For example, in an exec with Common Lisp readtable, it correctly completes symbols that start with \\, or a mixed-case symbol written with vertical bars. Also, Escape completion computes character widths correctly when it lowercases an upper case string, rather than leave some garbage bits on the display.
- The off-by-the-descent bug wherein TTYIN sometimes left stray bits at the bottom of the window has been fixed.

Section 26.4.3 Display Editing Commands

(III:26.25)

?= and Meta-P no longer hang if you had an unbalanced string quote in the input.

?=, Meta-P, and the **FIX** command now work correctly when there are NS characters in the input.

The printout for **?=** is now improved; it respects *print-case*, matches up keywords better, and prints abstract syntax descriptions (such as for cl:do) a bit more clearly.

SMARTARGLIST fetches the argument lists of cl:compiled functions, so **?=** now works in more cases.

The Ctrl-X command, when the caret is already positioned at the end of the input and everything but parentheses are balanced (i.e., no unbalanced string quotes or vertical bars), types as many closing parentheses as necessary to complete the input and then returns, much as if you had typed right bracket ("]") in Interlisp. Thus, if the cursor is somewhere in the middle of the input, typing two Ctrl-X's is sufficient to complete (assuming all you needed to type were some more parens).

TTYIN can now be used as a substitute for PROMPTFORWORD. The new function TTYINPROMPTFORWORD takes the same set of arguments as PROMPTFORWORD. In the most common cases it then calls TTYIN in "promptforword" mode, so that you can use the mouse and other TTYIN commands on the input. For cases it can't handle, it calls the old PROMPTFORWORD. These cases are: DONTECHOTYPEIN.FLG or KEYBD.CHANNEL is non-NIL; ECHO.CHANNEL is not a displaystream; or TERMINCHARS.LST contains a character other than cr, space or ^X and you have set the variable TTYIN.USE.EXACT.CHARS (initially NIL) to T. TTYIN saves the old definition of PROMPTFORWORD, so you can either have your program explicitly call TTYINPROMPTFORWORD instead of PROMPTFORWORD, or you can have all calls to

PROMPTFORWORD changed by doing a (MOVD 'TTYINPROMPTFORWORD 'PROMPTFORWORD).

Section 26.4.5 Useful Macros

(III:26.29)

CTRLUFLG is no longer supported by default. To use this feature, turn it on explicitly: **(INTERRUPTCHAR (CHARCODE ^U) 'CTRLUFLG)**.

Chapter 27 Graphic Output Operations

Section 27.1.3 Bitmaps

Note: The printed representation of bitmaps has changed. Please see release notes Chapter 3, Integration of Interlisp-D/ Common Lisp, "Bitmap Syntax."

(III:27.4)

The following function has been added to Bitmap Operations between the functions **EXPANDBITMAP** and **SHRINKBITMAP**:

(**ROTATE-BITMAP** *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.

(III:27.4)

In the Medley release, the **EDITBM** function is substantially faster with the inclusion of **FASTEDITBM** (a former LispUsers module) in the sysout.

Section 27.3 Accessing Image Stream Fields

The following functions were not documented in the Koto release of the *Interlisp-D Reference Manual*:

(**DSPCLEOL** *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** *DS CHAR X Y TTBL*) [Function]

Backs up over character code *CHAR* in the display stream *DS*, 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.

Section 27.6 Drawing Lines

(III:27.17)

The non-**NIL** value of the *DASHING* argument of **DRAWLINE** uses LINEWITHBRUSH. LINEWITHBRUSH is a width-by-width brush which draws then lifts.

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).

(III:27.18)

The **RELDRAWTO** function has been corrected so that it no longer draws a spot if the *DX* and *DY* arguments are 0.

Section 27.7 Drawing Curves

(III:27.18)

For the brush width value of **NIL,** the previous default value **(ROUND 1)** has been changed. The default value for the brush width value **NIL** is the **DSPSCALE** of the stream (that is, 1 printer's point wide).

(III:27.19)

A new image stream function, **DRAWARC**, follows **DRAWCIRCLE** in the *InterLisp-D Reference Manual*.

(**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**.

Section 27.8 Miscellaneous Drawing and Printing Operations

(III:27.20)

To have a filled polygon print correctly, set the global variable **PRINTSERVICE** to floating point value 9.0 for printers running Services 9.0 or later.

When using **FILLPOLYGON** to be sent to Xerox 8044 Interpress printers, the global variable **PRINTSERVICE** must be set to the same value as the Print Service installed on your printer, currently either 8.0, 9.0 or 10.0. Thus, if your printer is running Print Service 9.0, you must set the global variable **PRINTSERVICE** to the floating

point value 9.0. This works around an incompatible change in the Xerox 8044 Interpress implementation.

In Medley, Interpress curves are now rendered at a lower accuracy, allowing faster hardcopy. The spline is now rendered at 1/150 inch; in Lyric it was 1/300 inch.

The following function was omitted from previous version of the *Interlisp-D Reference Manual*:

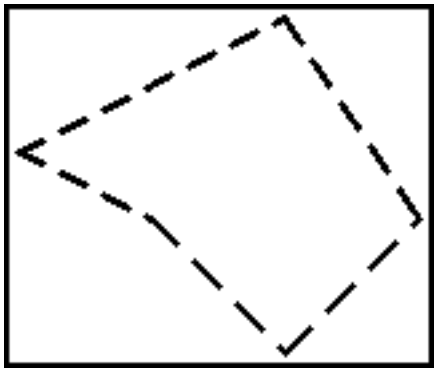
(**DRAWPOLYGON** *POINTS CLOSED BRUSH DASHING STREAM*) [Function]

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)
```

would draw a polygon like the following on the display stream XX.



(III:27.20)

The function **FILLPOLYGON** contains two new arguments, *OPERATION* and *WINDNUMBER*. The new form for the function, and definitions for added arguments, follow.

(**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 '((125 . 125) (150 . 200) (175 . 125)
               (125 . 175) (175 . 175))
             GRAYSHADE WINDOW)
```

would produce a display something like this:

This fill convention also takes into account all polygons in *POINTS*, if it specifies multiple polygons.

Section 27.12 Fonts

A revised set of font printing metrics is a part of the Lyric release of Lisp. Note that Koto font files are still available to users who request them.

With the revised font set the interline spacing (line leading) is now consistent across all fonts within a point size. Previously, text with multiple fonts (but with the same point size, i.e., if a word were made bold or italic, or if the family were changed) would have different leading on different lines. The new .WD files clean up document appearance.

Note that these printer metric changes affect only hardcopy, not the display. The contents of the display fonts are essentially unchanged in Lyric.

Generally, line leading in the Lyric font files is tighter than in previous releases of the fonts. The default line leading is now the same as the font's nominal point size. As a consequence of the above, any text file (one not already formatted for Interpress) which is printed after installation of the new fonts will be formatted to a different length. This means that decisions regarding TEdit line leading, widows and orphans, left/right pages, references to page numbers, etc. will need to change. Koto documentation produced by users may need to be reformatted with different line leading, using the new fonts.

All of the font files now have a new naming scheme, which allows **FONTSAVAILABLE** to be able to do more accurate pattern matching. For example, the display font file for modern 8 bold italics used to be named:

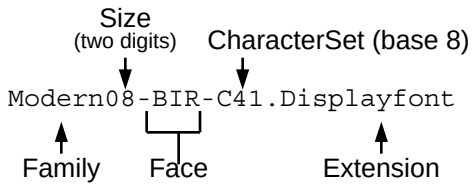
```
Modern8-B-I-C41.Displayfont
```

The file is now named:

```
Modern08-BIR-C41.Displayfont
```

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`).



The old file naming convention is still supported, however, with the exception of the old Strike file naming convention. In Lyric, **FONTCREATE** will first search for fonts using the new font naming convention, and if the desired font is not found it will search using the Koto convention.

Compatibility considerations You can continue using the old printer metrics (.WD files) in Lyric, thus preserving document looks between Koto and Lyric. If you choose to do so, it is recommended that you rename your old .WD files to the new naming scheme (see above), so that you benefit from the changes to the font searching mechanisms. However, we strongly urge you to use the new .WD files. Otherwise, if you exchange TEdit documents with a site that is using the new files, the documents will print differently at the two sites. The creation date, rather than the naming convention, determines whether a .WD file represents the old or new format.

If, after installing the new .WD files, you wish to print a document using the old Koto formatting, make the font variable **INTERPRESSFONTDIRECTORIES** point to a directory containing the Koto font files. Also any Lyric printer font file information must be uncached from the sysout. To uncache the fonts, perform

```
(for INFO in (FONTSAVAILABLE '* '* '* '*
                    'INTERPRESS)
   do (APPLY 'SETFONTDESCRIPTOR INFO))
```

(III:27.30)

(**STRINGWIDTH** *STR FONT FLG RDTBL*) [Function]

In Lyric **STRINGWIDTH** observes ***PRINT-LEVEL*** and ***PRINT-LENGTH***.

In Medley, STRINGWIDTH with a NIL argument no longer returns the string width of the string with *STANDARD-OUTPUT* font. It now uses DEFAULTFONT.

Some new font manipulation functions have been added to Lisp. They are:

(**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)
```

writes a font file which is identical in appearance to the current state of Gacha 10 charset 0.

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.

Section 27.13 Font Files and Font Directories

(III:27.31)

Press fonts are not part of the sysout since PRESS is now a Library module.

Section 27.14 Font Classes

(III:27.32-27.48)

This section has been expunged from the *InterLisp-D Reference Manual*. Renumber the sections which followed the old Section 27.14 as

SECTION 27.15 ⇒ SECTION 27.14 Font Profiles

SECTION 27.16 ⇒ SECTION 27.15 Image Objects

SECTION 27.17 ⇒ SECTION 27.16 Implementation of Image Streams

Section 27.14 Font Profiles

(III:27.34)

The variable **FONTCHANGEFLG** has an additional value, **ALL**. **FONTCHANGEFLG=ALL** indicates that all calls to **CHANGEFONT** are executed.

(III:27.33-34)

The function **FONTNAME** no longer exists. This function was previously used in Interlisp-D to collect the names and values of variables on **FONTDEFSVARS**. The variable **FONTDEFSVARS** is no longer used; it was appropriate when most output devices were fixed-pitch, "line-printer" style devices, but is not suitable for use when most output devices are laser printers.

Chapter 28 Windows and Menus

Section 28.4 Windows

(III:28.13, 28.38)

The **ADDMENU** function will change a window's **RESHAPEFN** and also will change the window's **REPAINTFN**.

Section 28.4.5 Reshaping Windows

(III:28.17)

The Lisp window system allows the following minimum window sizes:

When creating a new window, the width and height specified must be at least 9, or else you will get an error "region too small to use as a window"

When reshaping a window, the smallest shape you can get is width = 26 and height = height of the font to be used in the window. If you specify a smaller region, **SHAPEW** will simply adjust it to fit these limits.

Section 28.4.8 Shrinking Windows Into Icons

(III:28.22)

SHRINKFN [Window property]

In previous releases, there was a bug in the attached window system such that if an attached window had a **SHRINKFN** of the single symbol DON'T, attempting to shrink the window resulted in a break with the message "UNDEFINED FUNCTION DON'T." For this case in Lyric, all windows that can be shrunk will be, while those windows with a **SHRINKFN** of the symbol DON'T will be left open.

To facilitate the management of window regions, the window property **EXPANDREGIONFN** has been added to Lisp. This feature allows applications to arrange for reshaping a window when it is expanded.

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**).

(III:28.23)

Add the variable **DEFAULTICONFN** to the Icon section of the *InterLisp-D Reference Manual*:

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. It is described further in Appendix B (ICONW).

(III:28.23)

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))
```

Section 28.4.11 Terminal I/O and Page Holding

(III:28.29)

TTYDISPLAYSTREAM has been fixed so that it can be successfully used with non-windows.

Section 28.5 Menus

Two features have been added to this section, ICONW for creating icons, and FREE MENU, for creating and using free menus. Both features were formerly part of the Lisp Library.

The description for ICONW is in Appendix C. The FREE MENU description is in Appendix D.

The Lyric version of Free Menu differs in some respects from the Koto version of Free Menu. Following is a description of the incompatible feature changes from the old version to the new version of Free Menu. Some of the terminology used in these

notes is introduced in the Free Menu documentation found in Appendix B. Please reference Appendix B before reading the following notes.

- The function **FREEMENU** is used to create a Free Menu, replacing and combining the functions **FM.MAKEMENU** and **FM.FORMATMENU.**

The description of Free Menu has these changes:

1. There is no longer a **WINDOWPROPS** list in the Free Menu Description. Instead, the window properties **TITLE** and **BORDER** thatwere previously set in the **WINDOWPROPS** list can now be passed to the function **FREEMENU**. Other window properties (like **FM.PROMPTWINDOW**) can be set directly after Free Menu returns the window using the system function **WINDOWPROP**. See Appendix B, Section 28.7.14, Free Menu Window Properties.

2. 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 has been modified as follows:

1. **3STATE** items now have states **OFF**, **NIL**, and **T** (instead of a **NEUTRAL** state). They appear by default in the **NIL** state.

2. **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.

3. **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.

4. **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.

When you start editing with the right mouse button the item is first cleared.

5. **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.

6. **TITLE** items are replaced by **DISPLAY** items, which work the same way.

With Free Menu, the item interface functions can take the actual item datatype, the item's *ID* or *LABEL*, or a list of the form (**GROUPID** *ITEMID*) specifying a particular item in a group, as the *ITEM* argument.

The description for ICONW is in Appendix B. The FREE MENU description is in Appendix C.

These changes have occurred in the Free Menu Interface functions:

(**FREEMENU** *DESCRIPTION TITLE BACKGROUND BORDER*) [Function]

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*) [Function]

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*) [Function]

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*) [Function]

Has a 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*) [Function]

Has a 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*) [Function]

Replaces **FM.FIXSHAPE**

(**FM.HIGHLIGHTITEM** *ITEM WINDOW*) [Function]

Replaces **FM.SHADEITEM** and **FM.SHADEITEMBM**.

FM.HIGHLIGHTITEM will programmatically highlight an item, as specified by its **HIGHLIGHT** 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**.

Section 28.6.2 Attached Prompt Windows

(**GETPROMPTWINDOW** *MAINWINDOW #LINES FONT DONTCREATE* [Function]

In the Lyric release, the prompt window created by **GETPROMPTWINDOW** is *not* independently closeable, as it was in Koto. That is, selecting **Close** from the right-button window menu in the prompt window is the same as selecting it from the menu of any other window in the group—the entire window group is closed.

Section 28.6.3 Window Operations and Attached Windows

(III:28.51)

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**.

Chapter 29 Hardcopy Facilities

(III:29.3)

The **HARDCOPYW** function now has an additional argument, *HARDCOPYTITLE*, which allows you to change or eliminate the

"Window Image" message on IP screen images. Moreover, **HARDCOPYW** function now allows you to print large images occupying more than one page.

(**HARDCOPYW** *WINDOW/BITMAP/REGION FILE HOST SCALEFACTOR ROTATION PRINTERTYPE HARDCOPYTITLE*) [Function]

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.

Chapter 30 Terminal Input/Output

Section 30.1 Interrupt Characters

(III:30.2)

Control-P The Control-P (**PRINTLEVEL**) interrupt is no longer supported. The interrupt of that name still exists and is defaultly assigned to Control-P, but has no effect on printing.

Control-T The Control-T interrupt flashes the window belonging to the tty process and prints its status information in the prompt window. This avoids disrupting the user typescript.

(III:30.3)

(**INTERRUPTCHAR** CHAR TYP/FORM HARDFLG —) [Function]

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).

Section 30.2.3 Line Buffering

(III:30.11-12)

The **BKSYSBUF** function has been changed, for compatibility reasons. The description now reads as follows:

(**BKSYSBUF** *X FLG RDTBL*) [Function]

BKSYSBUF appends the **PRIN1**-name of *X* to the system input buffer. The effect is the same as though the user had typed *X*. 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.

(III:30.12)

Add the function **BKSYSCHARCODE** used in line buffering:

(**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**.

Section 30.4.1 Changing the Cursor Image

(***III:30.14***)

The **CURSOR** record has been changed to a DATATYPE, and its field names have changed in the following way:

Old Field Name	New Field Name
CURSORBITMAP	**CUIMAGE**
CURSORHOTSPOTX	**CUHOTSPOTX**
CURSORHOTSPOTY	**CUHOTSPOTY**

The **CURSORHOTSPOT** field no longer exists; its value can be fetched by composing **CUHOTSPOTX** and **CUHOTSPOTY** into a **POSITION**, or stored by destructuring a **POSITION** into those fields.

In Lyric, the **CURSORCREATE** function accepted as its argument bitmaps of any size, but caused an obscure error. In Medley, a bitmap that is bigger than 16 high or 16 wide will cause an ILLEGAL ARGUMENT error.

Section 30.5 Keyboard Interpretation

(***III*:*30. 19–20***)

(**KEYDOWNP** *KEYNAME*) [Function]

(**KEYACTION** *KEYNAME ACTIONS* —) [Function]

KEYNAME is interpreted differently in Lyric: 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)
```

Note: The key labeled HELP on the 1186 is named DBK-HELP for use in KEYACTION.

Section 30.6 Display Screen

(III:30.22-23)

(**CHANGEBACKGROUND** SHADE —) [Function]

The function **CHANGEBACKGROUND** treats the *SHADE* argument as a 4 X 4 texture. The **CHANGEBACKGROUNDBORDER** function, on the other hand, treats the *SHADE* argument as a 2 X 8 texture.

Therefore, note that the same *SHADE* argument, when used by the two functions, will not necessarily produce the same background and border shades on the display screen.

(III:30.23)

The **VIDEORATE** function works only on the 1108. Append the following note to the **VIDEORATE** function description:

(**VIDEORATE** TYPE) [Function]

Note: **VIDEORATE** does not work on the 1186.

Section 30.7 Miscellaneous Terminal I/O

(III:30.24)

(**BEEPON** FREQ) [Function]

The argument *FREQ* is measured in hertz, not in **TICKS**.

Chapter 31 Ethernet

Section 31.3.1 Name and Address Conventions

(III:31.8-9)

Amend the first paragraph, describing **NSADDRESS**, to list, in order, the components of **NSADDRESS:**

Addresses of hosts in the NS world consist of three parts, a network number, a machine number, and a socket number. These three parts are embodied in the Interlisp-D data type **NSADDRESS**. The components of **NSADDRESS** are 32-bit network, 48-bit host, 16-bit socket.

Move the following sentence from page 31.9 of the *IRM* to the last paragraph of Name and Address Conventions on page 31.8:

If you wish to manipulate **NSADDRESS** and **NSNAME** objects directly you should load the Lisp Library Module ETHERRECORDS.

NS Address Format

In Medley, you can now specify NS addresses in decimal notation, the form presented by the Chat interface of Network Services software. In this notation, a decimal number is broken up by hyphens every 3 digits, much like commas in standard American numerical notation. You can also specify a full 48-bit host number in octal without breaking it into 16-bit segments.

An NS address is specified in the form:

net#host#socket

If the address contains a hyphen in any field, the entire address is interpreted in decimal; otherwise in octal. The field *socket* is optional, and is defaulted appropriately for the application; if specified, it is a single integer in the same radix as the rest of the address. The field *net* and its terminating # are optional, defaulting usually to the directly-connected network. The fields *net* and *host* are non-negative integers written in one of three forms:

• A sequence of 16-bit octal numbers, separated by periods.

• A single integer in octal radix.

• A sequence of 3-digit decimal numbers, separated by hyphens.

The special variable ***NSADDRESS-FORMAT*** specifies the form used whenever the system prints an NS address object. Its possible values are:

NIL Octal radix, with the host number in three 16-bit parts, the same as in Lyric.

:OCTAL Octal radix without separators.

:DECIMAL Decimal radix with hyphens.

For example, the following all represent the same address, in the three formats listed above:

```
1750#0.125000.76771#

1750#25200076771#

1-000#2-852-158-969#
```

The following functions exist for manipulating NS addresses:

(PARSE-NSADDRESS *STR DEFAULTSOCKET)* [Function]

Parses the string *STR* into an NS address by the rules listed above, or returns NIL if *STR* is not a well-formed address. If *DEFAULTSOCKET* is non-NIL and the string does not include a socket field, the socket of the resulting NS address is set to *DEFAULTSOCKET.*

(COERCE-TO-NSADDRESS *HOST DEFAULTSOCKET)* [Function]

Returns an NSADDRESS object corresponding to *HOST*, or NIL if it can't. This function should be called by any software wanting to convert a user-supplied NS host specification into a network address. *HOST* can be any one of the following:

• The name of a host, whose address is found by consulting the Clearinghouse data base.

• A symbol or string in the syntax of an NS address, as described above.

• An NSADDRESS object.

• A list of the form (NSHOSTNUMBER *a b c)*, specifying the host number as three 16-bit values. In this case, the network number is omitted (zero).

If *DEFAULTSOCKET* is non-NIL and the socket is unspecified, the socket of the result is set to *DEFAULTSOCKET* (if *HOST* is an NSADDRESS object, it is copied in this case).

Network Routing Maintenance

The representation of Pup and NS routing tables has changed, and the background gateway listener processes have been tuned to significantly reduce their overhead.

The INFO command for the Pup and NS gateway listener processes in the process status window now display their routing tables. Clicking with the left button displays the table in random order, middle button displays the table sorted by network number (this takes a little longer).

Section 31.3.2 Clearinghouse Functions

(III:31.9)

The variable **AUTHENTICATION.NET.HINT** has been added to Clearinghouse Functions. It follows the **CH.NET.HINT** variable in the *Interlisp-D Reference Manual*.

AUTHENTICATION.NET.HINT [Variable]

AUTHENTICATION.NET.HINT can be set to **CH.NET.HINT** to speed up the initial authentication connection. Its value is interpreted in the same manner as **CH.NET.HINT**.

Section 31.3.3 NS Printing

(III:31.12)

With the Medley release there is now a single Printer Watcher process for all NS printers. This mean you won't get a stack overflow if you hardcopy many files in quick succession.

Section 31.3.5.3 Performing Courier Transactions

(III:31.20-21)

The **COURIER.OPEN** function requires that a courier server be running on the host machine.

Section 31.3.5.3.3 Using Bulk Data Transfer

(III:31.24-25)

The following is a correction and clarification to the description in the *Interlisp-D Reference Manual* of receiving values from a bulk data transfer:

It is possible for a Courier procedure to return both bulk data, in the form of a bulk data sink, and a single value (or list of values) as the normal result of the call. However, the Lisp function **COURIER.CALL** only returns one value, either the bulk data stream (when the bulk data sink argument is NIL) or the regular value. There are two principal ways in which a caller can obtain both values.

The usual way to get both values is to pass a function as the bulk data argument, have it retrieve the bulk data and prcoess it as a side-effect (e.g., store it into a variable bound around the **COURIER.CALL**), then return NIL so that the procedure's returned value is returned from **COURIER.CALL**.

The other way, which is documented incorrectly in the *IRM*, is to pass NIL as the bulkdata argument, thus getting the bulk data stream back from **COURIER.CALL**, process the stream, and then get the procedure's returned value by closing the stream. Contrary to the *IRM*, however, you have to close the bulk data stream using its internal close function, **SPP.CLOSE**, rather than the user-level function **CLOSEF**, which consumes the value internally and returns only the stream.

Section 31.5 Pup Level One Functions

\10MBTYPE.PUP [Variable]

\10MBTYPE.3TO10 [Variable]

The values of these variables are the 10MB Ethernet encapsulation types for PUP packets and Pup-to-10MB address translation packets, respectively. The initial values of these variables are 512 and 513, respectively. However, these values are illegal for an Ethernet conforming to IEEE 802.3 specifications.

New encapsulation types have been defined for IEEE 802.3 networks. To use them, set the variable **\10MBTYPE.PUP** to 2560 (decimal) and **\10MBTYPE.3TO10** to 2561. Then call either **(RESTART.ETHER)** or **(LOGOUT)**, so that the Ethernet code can reinitialize itself. It may be convenient for a site to smash these values directly into the standard sysout everyone fetches by using the function **READSYS** and its `^V` command from the TeleRaid Library module (the sysout must be on disk or a random-access file server). Note that *all* pup hosts on a network (servers as well as workstations) must simultaneously choose to use the new values; those using different values will be unable to communicate with each other. The System Tool must also be upgraded at the same time.

Section 31.6.1 Creating and Managing XIPs

The function NSNET.DISTANCE was previously undocumented.

(**NSNET.DISTANCE** *NET#)* [Function]

Returns the "hop count" to network *NET#,* i.e., the number of gateways through which an XIP must pass to reach *NET#,* according to the best routing information known at this point. The local (directly-connected) network is considered to be zero hops away. Current convention is that an inaccessible network is 16 hops away. **NSNET.DISTANCE** may need to wait to obtain routing information from an Internetwork Router if *NET#* is not currently in its routing cache.

[This page intentionally blank]

5. LIBRARY MODULES

This section contains release notes indicating changes that have occurred in the library modules since the Lyric release. Medley changes are indicated with revision bars in the right margin.

Refer to the *Lisp Library Modules* manual, Medley release, for complete documentation of the library modules.

Modules that are New, Moved, or Replaced

Modules Moved from the Library to LispUsers

Big
BitMapFns
BusExtender
BusMaster
CirclPrint
CheckSet
CompileBang
Color
C150Stream
DECL
DInfo
FileCache
HelpSys
Iris
LambdaTran
PCallStats
ReadAIS

Modules Moved from LispUsers to the Library

Cash-File
Hash-File
SysEdit
TableBrowser

Modules Moved to Their Own Manuals

TEdit
Sketch
CML, CMLArray, CMLArrayInspector (part of Xerox Common Lisp)

Modules Moved From the Sysout Into the Library

DEdit
Masterscope
Match

Press

Modules Moved From the Library Into the Sysout

IconW
FreeMenu

Modules Replaced

Old: FX-80stream, FastFX-80stream, FXprinter
New: FX-80Printer

Old: WhereIs
New: Where-Is

New Modules

SysEdit
TableBrowser
TextModules

Details of Changes

4045XLPStream

Enabled its graphics capabilities; added 1108 cable/connector pin-outs.

A new function has been added to allow owners of the international 4045 (non-USA model) to use the 4045XLPStream software.

(4045XLP.CHANGE.MODE *MODE*) [Function]

This function changes the internal parameters of the software to allow printing on A4 paper with the international fonts. *MODE* is a string, either "USA" or "INTERNATIONAL", with the default being "USA". Do not use this function unless you have the international font set and A4 paper tray on a non-USA 4045. A4 page size is 2475 pixels wide by 3525 pixels high in portrait, and 3525 x 2475 in landscape mode.

Cash-File

The new library module Cash-File was formerly in LispUsers. Cash-File is a front end to Hash-File which uses a hash table to cache accesses to hash files. This can provide a significant performance improvement in applications which access a small number of keys repeatedly. For example, the Where-Is library

module uses this module to achieve acceptable interactive performance.

Centronics

Added cable/connector pin-out.

Chat

Added information about EMACS.

CopyFiles

When told to copy to a non-existent NS subdirectory, it now asks if it should create it.

DataBaseFns

Clarifications in the documentation of LOADDBFLG and SAVEDBFLG are included in Medley.

EditBitMap

Added a description of its user interface.

FileBrowser

Added enhanced features to Load, Compile, Edit; it now preserves path name of source files when copying to another machine or user; sorts files by attributes; and prints hard copies of directory listings.

The FB command now ignores the package of the attributes you optionally specify, so you can easily use it from a non-Interlisp exec.

The enclosing *'s are now included with the names of the variables *EDITMODE* and *DEFAULT-CLEANUP-COMPILER* .

In addition to having outstanding problems fixed, FileBrowser has several new features and NS enhancements.

New features:

- There is an Abort button available during any operation of indefinite duration.
- You can scroll or reshape a FileBrowser that is "busy", e.g, while doing a Recompute.
- The browser title includes a timestamp of when the browser contents were last Recomputed.
- There is a new subcommand of See, "FileBrowse", which opens a FileBrowser on the selected subdirectory. This replaces the odd functionality of the old See/Edit commands that assumed

that any file with null name and extension must be a directory; those commands now always treat the selection as a file. FileBrowse is mainly useful in the following situations:

- When browsing NS directories with depth set finite, or when browsing the top level of a server, which is automatically depth 1.

- When browsing on Unix, a device that gives Lisp no indication of whether a filename is a directory or not.

• There is a subcommand of Recompute, "Set Depth" that can be used to set the enumeration depth for future recomputes and recursive FileBrowses. You can also set the depth in an FB command by appending :DEPTH n to the command line, e.g., FB "{Pogo:}<Carstairs>" :DEPTH 1.

The depth counts levels of directories below the last directory in the pattern not containing a wildcard; depth 1 means just the immediate descendants of that directory. Depth is ignored for nontrivial patterns, i.e., anything but "*.*".

• Another new subcommand of Recompute, "Shape to Fit", widens or narrows the browser so that all fields, and no more, are visible but not wider than the screen.

• Directory items are now displayed like files, e.g., you'll see a single line

Lisp>

rather than the double line

<Carstairs>Lisp>

NIL

In addition, the "page" count for a directory item is now the total page size of the directory subtree rooted there.

• FileBrowser consumes somewhat less storage now, and there have been some performance improvements, especially for very large browsers.

FTPServer

FTPServer now supports DELFILE.

The Medley release fixes several bugs in Lisp's handling of PUP FTP connections relating to password handling and filename recognition.

FX-80Driver

New software, new text, and 1108/1186 cable/connector pin-outs have been added.

Comments are now printed in a compressed font .

GCHax

Documentation contains a new description of the STORAGE function.

Grapher

Grapher can now print graphs larger than one page. The variable GRAPH/HARDCOPY/FORMAT is used to control the format of the graph when printing to paper. See the function HARDCOPYGRAPH and the variable GRAPH/HARDCOPY/FORMAT in the documentation for Grapher for more information.

A new GRAPH.PROPS field has been added to Graph record, which produces a list in property-list format, and is accessed by the function GRAPHERPROP.

Hash

Hash is provided for backwards compatability. New applications should use the Medley library module Hash-File instead of this module.

Hash-File

Hash-File is a new library module, upgraded from LispUsers. Hash-File is similar to but not compatible with the Lyric library module, Hash. Hash-File is modeled after the Common Lisp hash table facility, and Hash was modeled after the Interlisp hash array facility.

Kermit

Reference to an excellent text/reference book has been added.

MasterScope

Break when graying a browser has been fixed.

In Medley, MasterScope .LCOM files have been changed to .dfasl file extensions. MasterScope now recognizes Common Lisp structures.

NSMaintain

The module NSMaintain has been completely revised and has all new documentation. Most commands auto-complete on one or two keystrokes. The Change Password command works again, and there are several new commands for listing objects in the Clearinghouse data base and for manipulating the access lists of groups. There is a more rational set of default inputs offered for most commands, and better feedback is given as to whether a command succeeded or failed.

RS232

The RS232.TRACE function is now documented in the Medley release.

Spy

This version of SPY library module works better with Common Lisp and incorporates several new features:

- Enters the pending mode when you bring up the SPY menu by pressing the left or middle button while the control key is down. Any action invoked from the menu is deferred until you next press the left or middle mouse button. For example, you can delete several nodes and then do one update.
- Keeps track of non-symbol frame names.
- Shows the package prefix of symbols in the display.
- Invokes "Merge" menu item from a node menu allowing for a node to merge with its caller.
- Updates SPY.NOMERGEFNS to correspond more closely to "system" functions in Medley.
- Knows about the Medley interpreter.

TableBrowser

- New functions TB.UNSELECT.ITEM and TB.UNSELECT.ALL.ITEMS fill an inadvertant void in the Lyric version.
- Several off-by-ones in the display algorithms have been fixed.
- Performance on large browsers is improved.
- Clarification of TBAFTERCLOSEFN documentation is included in the Medley release.
- New options to TB.MAKE.BROWSER:
  - The option LINESPERITEM, previously documented but not implemented, is now supported. Alternatively, you can specify explicitly the height of items by giving the options ITEMHEIGHT (total height of each item) and/or BASELINE (the height of the "baseline" relative to the bottom of the item; zero if you don't set it). The BASELINE is used for two things: (1) the *ypos* of the window is set there when the browser's print function is called, and (2) selection and deletion marks are centered between the baseline and the top of the item. Specifying LINESPERITEM is a shorthand method for setting ITEMHEIGHT to fontheight*#lines and BASELINE to fontheight*(#lines-1)+fontdescent (i.e., font's baseline for the first line of the item), so that the selection marker, deletion lines, and positioning for printing all point at the first line of a multi-line item. One further difference: if you change the font of the browser, TableBrowser will recompute the height and baseline parameters if you specified LINESPERITEM, but not if you specified ITEMHEIGHT.

- You can specify an auxiliary window that is to be horizontally scrolled in parallel with the main window by giving the window as the HEADINGWINDOW option. The WIDTH of the window's EXTENT property is maintained in synch with main window. You still need to create the auxiliary window, attach it where you want it and supply it with a REPAINTFN. This is how FileBrowser implements its header line consisting of "Name" and the attribute names.

- The option LINETHICKNESS specifies how thick to draw deleion lines. It defaults to TB.DELETEDLINEHEIGHT, initially 1. Making it the height of an item gives an alternative "total blackout" method of deletion.

TCP-IP

Added revised/expanded installation procedure.

DIR to VMS via TCP now works.

TCP Chat hosts can now be lowercase.

(TCPFTP.SERVER) now spawns process and runs in it .

TCP-IP to a Sun returns the top-level directory.

TCPFTP.DEFAULT.FILETYPES now contains correct entries for LCOM, lcom, DFASL, and dfasl.

Files loaded by the high-level modules TCPFTP, TCPFTPSRV, TCPCHAT, and TCPTFTP automatically load their dependencies. If you load files by hand, you must also load their dependencies first. See the section "File Dependencies," or the TCP-IP documentation for more information.

There is a new flag:

TCP.ALWAYS.READ.HOSTS.FILE [Variable]

This flag is initially T. Setting it to NIL will cause the system to parse the hosts.txt file only when the filename (stored in the configuration file) is different from the previously read filename, or the write date of the file has changed. The hosts.txt file will always be read at least once when loading the software into a clean sysout.

If you change your IP.INIT file while TCP-IP is running, you will be prompted to confirm **Restarting TCP**. In most cases, you should confirm the restart.

TExec

A TEXEC executive window no longer has GET in the menu of possible actions, since GETting text into an executive window makes no sense.

TextModules

TextModules is a new library module with the Medley release. It can be used to import and export textfiles and File Manager files. It can bring portable Common Lisp sources into the File Manager without losing any of their contents, and create new textfiles based on the File Manager's description of the textfile contents.

Virtual Keyboards

The Standard-Russian virtual keyboard now has uppercase Be (. . .) and Ve (. . .) in the right places.

Loading VirtualKeyboards now adds the item KEYBOARD to the default window menu as well as the background menu. Selecting this item from the default window menu allows you to specify a keyboard for an individual window.

Where-Is

Where-Is is a new library module, upgraded from LispUsers. This modules replaces the Lyric library module WhereIs. This is a new implementation of a facility similar to but not compatible with the Lyric library module WhereIs. Where-Is indexes all definers, but WhereIs only indexed Interlisp FNS definitions.

Additional Notes

DEdit is not error-protected. Doing a ^in a DEdit break window closes the DEdit window, too.

In addition, the modules work under all Lisp environments (Interlisp-D, Xerox Common Lisp, Common Lisp). However, many of the functions and variables used within the modules are those of Interlisp-D, and therefore you'll have to make sure that, when you are not in Interlisp, you use the IL: prefix.

Koto CML Library Module

If you have files that used the Koto CML library module, with its package-style symbol naming conventions, you will need to convert them to use the correct symbols in Lyric /Medley. The procedure is

briefly as follows: see the *Common Lisp Implementation Notes*, chapter 11, "Reader compatibility feature" for complete details on this mechanism:

First, set the global variable LITATOM-PACKAGE-CONVERSION-ENABLED to T. Then for each of your files, do

(LOAD *file* 'PROP)

(MAKEFILE *file* 'NEW)

Afterwards be sure to set the global variable LITATOM-PACKAGE-CONVERSION-ENABLED back to NIL.

[This page intentionally left blank]

A User's Guide to TEdit—Release Notes

For the Medley Release, TEdit has increased the number of expanded characters, added options to the **Put** and **Get** submenus, clarified several options in the Paragraph Looks and Page Layout menus, and added several minor items to the programmer's interface.

Expanded Characters

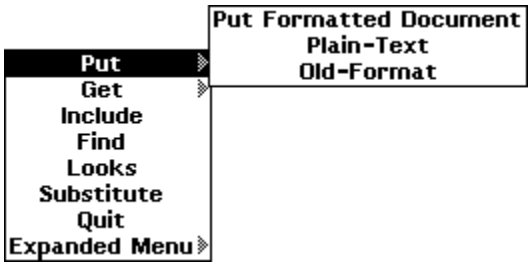
TEdit added the following abbreviations and expansions to the characters shown in Table 1.

Table 1. TEdit's Abbreviations and their Expanded Characters

Abbreviation	Expanded character name	Expansion Character
p	Pilcrow (proofreader's paragraph mark)	¶
t	Trademark	™
tm	Trademark	™
r	Registered trademark	®
1/3	Built-up fraction	⅓
x	Times sign	×
/	Division sign	÷
o (oh)	Degrees sign	°
L	Pound sterling sign	£
Y	Yen sign	¥
+	Plus-or-minus sign	±
^(shift-6)	Up arrow (NS character)	↑
ua	Up arrow (NS character)	↑
|	Down arrow (NS character)	↓
da	Down arrow (NS character)	↓
<-	Left arrow (NS character)	←
la	Left arrow (NS character)	←
_ (underscore)	Left arrow (NS character)	←
->	Right arrow (NS character)	→
ra	Right arrow (NS character)	→
=	Two-way arrows (NS characters)	⇔

Put Submenu

The drag-through menu for the **Put** command now has the following entries:

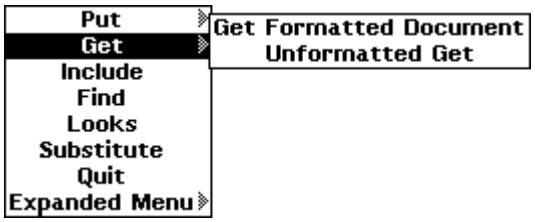


The **Put** command has a submenu that offers you several options for saving your file:

- Keep the formatting in the file. Use this **Put Formatted Document** option, which is the default, unless you have special requirements.
- Save the file as plain text, regardless of formatting. Using this **Plain-Text** option removes all of TEdit's formatting from the file, leaving plain text.
- Save TEdit files in an "old" format. This **Old-Format** option allows you save files in the format of a previous release of TEdit. This format option is provided for backward compatibility.

Get Submenu

The drag-through menu for the **Get** command now has the following entries:



Get has a submenu that offers you the option of retrieving a formatted file (**Get Formatted Document**), or retrieving a file as though it were plain text, with most formatting information appearing as black rectangles (•).

Clarified Paragraph Looks Menu Options

Both the menu options **New Page: Before After** and **Displaymode: Hardcopy** have expanded explanations.

New Page: Before After

Sometimes a page break occurs so that the first paragraph on a page is marked with the **Before** command. In these cases, the text flows continuously from the previous page to this page; a blank page does not appear between them.

Displaymode: Hardcopy

The Hardcopy displaymode command now works only when the text is printed in Interpress fonts.

Clarified Page Layout Menu Options

When specifying text to appear before or after page numbers, you can only enter text in the brackets; image objects are not allowed.

You may now specify a landscape page layout.

In the page layout menu, Modern 10 MRR is now the default page number font instead of Gacha 10. Also, there is a global variable, TEDIT.DEFAULT.FOLIO.LOOKS, that you can set to be any character-looks specification acceptable to TEDIT.LOOKS. The default (i.e., if you don't specify one in the page layout menu) is taken from there.

If you have set page formatting in the past, the page-numbering font has been set as well (even if you specified nothing). This behavior continues, but the default is more sensible, and can be changed.

You may now number the first page of a TEdit file 0 (zero).

TEdit now preserves text before and after page numbers after a file is saved.

Using numbers with decimal points in the "Text before page number" field in the page-layout menu now works properly.

Added Items to Programmer's Interface

The TEXTOBJ data structure has a correction, the TEDIT.INCLUDE, TEDIT.PARALOOKS and TEXTPROP functions are expanded, and the global variable TEDIT.KNOWN.FONTS is now documented.

Corrected the AFTERQUITFN Property

(AFTERQUITFN *WINDOW TEXTSTREAM*) is an optional user-supplied Lisp function that is called after ending an editing session to peform any required cleanup. The *WINDOW* argument was omitted in the manual.

Corrected the TITLEMENUFN Property

TEDIT.TITLEMENUFN is a window property, not a TEdit property as documented in the manual.

Corrected the TEXTOBJ Data Structure

The data structure called TEXTOBJ has as its first field of interest \WINDOW, not WINDOW as documented in the manual.

Expanded the TEDIT.INCLUDE Function

TEDIT.INCLUDE now accepts optional START and END arguments that instruct it to restrict its attention to a portion of the TEdit file, treating this part as a separate file. This feature is useful when you require that several distinct TEdit documents reside within a single TEdit file, for example, for database applications. Each document can be formatted and extracted separately.

CAUTION

If you use START and END arguments with INCLUDE, and then format the entire TEdit file, you will lose the formatting.

CAUTION

TEDIT.INCLUDE and OPENTEXTSTREAM take optional arguments that let you take a document out of the middle of a file. This option requires that Lisp be able to determine the length of the file before it is read. Some file protocols (TCP FTP in particular) don't let Lisp do this. If you try to use this option with a file that resides at the other end of a TCP connection (or, more generally, on any device where you cannot tell the length of the file until you have read the whole file), it won't work. The result will be that your document will contain no characters.

Expanded the TEDIT.PARALOOKS Function

TEDIT.PARALOOKS can be used to set NEWPAGEBEFORE, NEWPAGEAFTER, HARDCOPY, TYPE, SUBTYPE, REVISED, KEEP, STYLE, CHARSTYLES, and USERINFO parameters.

REVISED, if non-NIL, causes a revision bar to be printed one pica to the right of the right margin of the paragraph. It is a vertical bar 1 point wide from the top of the top line's ascent to the bottom of the bottom line's descent.

USERINFO can be used as a property list for saving information of interest to the user. It is generally used in a number of undocumented features (e.g. footnote support).

KEEP, STYLE, and CHARSTYLES are reserved for a future release.

Expanded the TEXTPROP Function

You can now also use your own properties, but these properties are not saved with the document if you **Put** it.

Added Documentation for a Global Variable

The following documentation should be added to TEdit's Global Variables.

TEDIT.KNOWN.FONTS [Variable]

A list of available fonts that appear in the Character Looks menu. The list in in the form ((name-in-the-menu-1 'Real-font-name-1) (name-in-the-menu-2 'Real-font-name-2) ...), for example, ((Classic 'CLASSIC) (Times 'TIMESROMAN)).

Changes to Programmer's Interface to TEdit

STREAM AND TEXTOBJ

All public TEdit functions (non- \) that take a *TEXTOBJ* argument accept either a TEXTOBJ or a text STREAM as that argument's value.

Changes, Additions and Corrections to TEdit functions

The function TEDIT.SINGLE.PAGEFORMAT is incorrectly documented in the Lisp Library. The following corrections should be noted: The arguments *PG#X*, *PG#Y*, and *PG#FONT* should be *PX, PY*, and *PFONT*, respectively.

The argument *PG#ALIGNMENT* should be *PQUAD*.

The order for the arguments, *TOP BOTTOM LEFT RIGHT* should be *LEFT RIGHT TOP BOTTOM*.

The argument *#COLS* should be *COLS*.

INTERCOLSPACE should be *INTERCOL*. And between the *INTERCOL* and *UNITS* arguments there is a *HEADINGS* argument.

The functions and its arguments look like:

(TEDIT.SINGLE.PAGEFORMAT *PAGE#S? PX PY PFONT PQUAD LEFT RIGHT TOP BOTTOM COLS COLWIDTH INTERCOL HEADINGS UNITS PAGEPROPS PAPERSIZE*) [Function]

PAGE#S? T if you want page numbers on this kind of page, else NIL.

PX The horizontal location of the page number, measured from the left edge of the paper. Negative values are measured from the paper's right edge.

PY The vertical location of the baseline for the page numbers, measured from the bottom of the paper. Negative values are measured from the top of the paper.

PFONT The font to be used to display the page numbers. This can be any specification that is acceptable to TEDIT.LOOKS.

PQUAD An atom that tells how the page number is to be aligned on the location specified by *PX* and *PY.* LEFT means the location is the lower-left corner of the page number. RIGHT means the location is the lower-right corner. CENTERED means the page number will be centered around the *PX* you specified.

LEFT The left margin—the distance from the left edge of the paper to the left edge of the first text column.

RIGHT The right margin—the distance from the right edge of the rightmost text column to the right edge of the paper.

TOP The top margin of the page—the distance from the top of the paper to the top of the first line of body text.

BOTTOM The bottom margin—the distance from the bottom of the last line of body text to the bottom of the paper.

COLS Number of columns (default is one).

COLWIDTH The column width (default is to evenly divide the available space among the #*COLS* columns).

INTERCOL The space between the right edge of one column and the left edge of the next column. Defaults to evenly divide the space left after the columns are set up. If there is more than one column, one or the other of *COLWIDTH* and *INTERCOLSPACE must* be specified.

HEADINGS A list of lists in the form of `((HEADINGNAME XLOCATION YLOCATION ) (HEADINGNAME XLOCATION YLOCATION ) . . .(HEADINGNAME XLOCATION YLOCATION )).`

UNITS The units used in setting the values you specified. May be one of the atoms PICAS, IN, INCHES, CM, POINTS. Default is POINTS.

PAGEPROPS A property list of extra information. Properties are STARTINGPAGE#, FOLIOINFO, and LANDSCAPE?.

STARTINGPAGE# is the first page's number; it is ignored if this isn't the first page.

FOLIOINFO is a list of information about page numbers, (FORMAT TEXTBEFORE TEXTAFTER). FORMAT can be one of ARABIC, LOWERROMAN, UPPERROMAN, or NIL (i.e., ARABIC). TEXTBEFORE is the text preceding the number, and TEXTAFTER is the text following the number.

LANDSCAPE? determines if the document is printed in the usual vertical format or printed in landscape format (horizontally). If NIL the document is printed vertically, if non-NIL the document is printed landscape. Defaults to NIL.

PAPERSIZE Is one of LETTER, LEGAL, the metric paper sizes (A0, A1, A2 A3, A4, A5, B0, B2, B3, B4), or NIL (which defaults to letter size).

TEDIT.GET accepts an open stream as the file to GET from. You may still pass it a TEXTOBJ, however.

(TEDIT.GET *STREAM FILE UNFORMATTED*?) [Function]

Performs the TEdit Get command, loading the text from *FILE* onto the editing stream *STREAM*—replacing the text that is being edited currently. If *FILE* is not supplied, the user will be asked for a file name. If *UNFORMATTED?* is non-NIL, *FILE* is treated as a plain-text document, and all of its contents are included—even TEdit formatting information.

You can now use TEDIT.PUT to store a TEdit document in the middle of a larger file (e.g., for saving TEdit documents as part of a database). The complete documentation is now as follows:

(TEDIT.PUT *STREAM FILE FORCENEW UNFORMATTED? OLDFORMAT?*) [Function]

Performs the TEdit Put command, saving the text from the text stream *STREAM* onto the file named *FILE.* If *FILE* is NIL, the user will be prompted for a file name. In this case, if *FORCENEW* is NIL, the user is offered the old file name as a default; if non-NIL, no default is given, forcing the user to specify a file name. If *UNFORMATTED?* is non-NIL, only characters are put in the file—no formatting. If *OLDFORMAT?* is non-NIL, the file will be written in the format used by the previous version of TEdit, for backward compatibility.

In order to store a TEdit document as part of another file, call TEDIT.PUT, passing an open stream on the file as the *FILE* argument. The stream should be open for output and positioned at the place you want TEdit to store the document (call this file pointer START). When TEDIT.PUT returns, the stream's end-of-file pointer will be just after the last byte in the newly-inserted document. Call this file pointer END. To subsequently retrieve the document from the middle of this other file, call OPENTEXTSTREAM on the file, passing the START and END pointers as the *START* and *END* arguments.

Note: When TEDIT.PUT returns, the stream will be open for INPUT.

The functions TEDIT.MOVE and TEDIT.COPY were not documented in Koto. They are:

(TEDIT.MOVE *FROM TO*) [Function]

FROM and *TO* are SELECTIONs. Moves the text described by *FROM* to the place described by *TO*, within the same text stream or between different text streams. The text described by *FROM* is deleted from its original location.

(TEDIT.COPY *FROM TO*) [Function]

FROM and *TO* are SELECTIONs. Copies the text described by *FROM* to the place described by *TO*, within the same text stream or between different text streams. The text described by *FROM* is not deleted in the *FROM* location.

Changes in the Documentation of TEdit Functions

The following functions have had the documentation of their arguments changed to reflect what will appear if you do a ?= or evaluate ARGLIST on one of these functions. Arguments that were corrected are indicated by bold italics (***arg***). Please note that what changed was the documentation, not the way the functions operate or the values of the arguments themselves.

(TEDIT.SETSEL *STREAM* ***CH#*** *LEN POINT* ***PENDINGDELFLG*** *LEAVECARETLOOKS OPERATION*) [Function]

(COERCETEXTOBJ *STREAM TYPE* ***OUTPUTSTREAM***) [Function]

(TEDIT.DELETE *STREAM* ***SEL*** *LEN*) [Function]

(TEDIT.INCLUDE ***STREAM*** *FILE START END*) [Function]

(TEDIT.FIND ***STREAM TARGETSTRING*** *START# END# WILDCARDS?*) [Function]

(TEDIT.GET.LOOKS ***STREAM CH#ORCHARLOOKS***) [Function]

(TEDIT.PARALOOKS ***STREAM*** *NEWLOOKS* ***SEL LEN***) [Function]

(TEDIT.COMPOUND.PAGEFORMAT ***FIRST VERSO RECTO***) [Function]

(TEXTOBJ ***STREAM***) [Function]

(TEXTSTREAM ***STREAM***) [Function]

(TEDIT.CARETLOOKS *STREAM* ***LOOKS***) [Function]

(TEDIT.NORMALIZECARET ***STREAM*** *SEL*) [Function]

(COPYTEXTSTREAM ***ORIGINAL*** *CROSSCOPY*) [Function]

(TEDIT.PROMPTPRINT ***TEXTSTREAM*** *MSG CLEAR?*) [Function]

(TEDIT.SETSYNTAX ***CHAR*** *CLASS TABLE*) [Function]

(TEDIT.GETSYNTAX ***CH*** *TABLE*) [Function]

(TEDIT.SETFUNCTION *CHARCODE FN* ***RTBL***) [Function]

(TEDIT.WORDGET ***CH*** *TABLE*) [Function]

(TEDIT.WORDSET ***CHARCODE*** *CLASS TABLE*) [Function]

(TEDIT.INSERT.OBJECT *OBJECT STREAM* ***CH#***) [Function]

The following functions were previously documented as accepting a TEXTOBJ. They all still take a TEXTOBJ but they will now also accept a STREAM as the first argument.

(TEDIT.FIND *STREAM TARGETSTRING START# END# WILDCARDS?*) [Function]

(TEDIT.GET.LOOKS *STREAM CH#ORCHARLOOKS*) [Function]

(TEDIT.PARALOOKS *STREAM NEWLOOKS SEL LEN*) [Function]

(TEXTSTREAM *STREAM*) [Function]

(TEDIT.NORMALIZECARET *STREAM SEL*) [Function]

(TEDIT.PROMPTPRINT *TEXTSTREAM MSG CLEAR?*) [Function]

New Features

For the benefit of NS file server users, TEdit now writes files of type TEDIT, instead of BINARY. As a result, LISTFILES and the FileBrowser are able to determine that the file is a TEdit file and call

TEdit to create the hardcopy. Previously, it was necessary that the TEdit file explicitly have the extension ".TEdit".

```
(OPENSTREAM file 'OUTPUT 'NEW '((TYPE TEDIT))).
```

This change is for formatted files only. Plain text files are still written as type TEXT. Also, on devices that don't support arbitrary file types (such as conventional mainframe file servers), the type TEDIT coerces to BINARY. Unfortunately, if you subsequently copy the file to an NS file server from such a device, the knowledge of its "true" file type is lost.

A User's Guide to Sketch—Release Notes

The Medley release of Sketch includes several new features, many added in response to user's requests. A programmer's interface allows sketches to be created by programs. This interface is described in a separate document (*The Programmer's Interface to Sketch*.)

Manipulating Sketch Elements

Adding and Deleting Control Points

Individual control points can now be added to and deleted from wires and curves.

Deleting Control Points

You now have the option to delete elements or delete a control point. Just select the **Delete** command, move the mouse cursor out through the grey arrow, then select the point to be deleted.

Defaults Command

Better Feedback for Creating Wires, Circles and Ellipses

Sketch now provides better feedback when you are creating circles, ellipses and wires. You are now prompted with an image of what the figure will look like if you release the left button. You can get the old feedback behavior (for example, if this is too slow) by selecting the **Feedback** subcommand from the **Defaults** submenu, then selecting the **Points only** subcommand from its submenu.

Arrowheads

A curved arrowhead shape was added and is now the default. Also, a command was added to the menu of arrowhead change operations that implements "look same" for arrowheads. To make the arrowheads on a collection of elements look the same: select **Change**; then, when prompted to select the elements to change, first select the element that has the desired arrowhead, then, in the same selection, add the elements that you want to look like the first one; then select the item **Arrowheads**, then the item **Both**, then the item **Same as First**.

Deleting Characters During Type-in

You can now delete characters by using the UNDO key, just as you would in TEdit. Type in a word or a phrase, then press the UNDO key, and the text will be deleted.

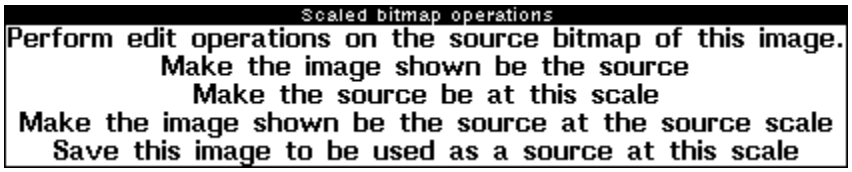
Using Bit Maps in a Sketch

Zooming Bitmaps

The bit image element provides a bitmap that zooms. Selecting the **Bit image** command from the command menu will prompt you for a region of the screen that will be inserted as a bit image into the sketch.

Changing Bitmaps

When you apply a **Change** command to a bit image that it is being viewed at actual size, you will be prompted with the same menu as a bitmap image object. If the image is being displayed at other than original scale, you will be given the menu shown below.



*Menu of commands offered when **Change** command is applied to a bit image that is not at the original scale.*

Freezing Sketch Elements

It is now possible to freeze elements, that is to make them unaffected by edit changes. Frozen elements will not have their control points highlighted (and hence cannot be selected) after an edit command has been selected. This provides a way to keep part of the figure fixed while editing on an overlapping part. It also reduces the number of control points. The **Freeze** command is a subcommand to the **Group** command. It will prompt you for a collection of elements that will then be frozen. Elements can be unfrozen by the **UnFreeze** command that is a subcommand to the **UnGroup** command.

Aligning Sketch Elements

Sketch contains a set of commands to align elements. The main menu command **Align** prompts for a collection of control points and moves them so that they all line up with the leftmost one.

Placing Multiple Copies of Elements

There is a new feature in Sketch that makes it much easier to place multiple copies of a collection of elements. While positioning the image of the elements during the **Copy** command, hold down the COPY key. A new copy of the elements will be positioned everytime a mouse button (left or right) is pressed and released, until either the image is placed completely outside the viewer or the COPY key is released before the mouse button is released.

Making the Window Fit the Sketch

The **Fit to window** subcommand under the **Move View** command will zoom the sketch so that it just fits within the current window. It

has a sub-subcommand **Fit window to sketch** that will reshape the window so that the entire sketch (at the size shown) just fits within it. This is useful if you change a sketch that was edited from a document.

Overlaying Figure Elements

Elements that have a filling property (boxes, text boxes, circles, polygons and closed curves) now have a mode property that determines how the filling should effect elements it covers. The option **Filling mode** now appears in the **Which aspect?** submenu.

Changing How Elements Overlap

Elements have an order in which they are displayed. An element that is displayed early can be covered by elements layed down later. Thus, changing the order in which overlapping elements are displayed can effect the resulting image. The **Bury** command provides three subcommands to change the order in which elements are displayed.

The **Bury** command will prompt you to select an element or elements and will change their order so that they are displayed first. That is, they will appear underneath any other elements. If you select more than one element, they will all be displayed before any non-selected elements and their relative order maintained. The **Send to bottom** subcommand does the same thing as **Bury**.

The **Bring to top** command is a subitem to the **Bury** command. It will prompt you to select an element or elements and will change their order so that they are displayed last. That is, they will appear on top of any other elements. If you select more than one element, they will all be displayed after any non-selected elements and their relative order maintained.

The **Reverse order** command is a subitem to the **Bury** command. It will prompt you to select a collection of elements and will reverse their display orders. A special case is when two elements are selected. In this case the element positions are switched.

Loading the Sketch Library Module in the 1186 Environment

The SKETCH executable files are too large to be contained on one floppy. The files are now distributed on two floppies: *Medley Library Floppy #3* and *Medley Library Floppy #4*. To load SKETCH, type the Interlisp exec command:

(FILESLOAD LOAD-SKETCH)

The LOAD-SKETCH function will copy all SKETCH files from #3; then prompt you to insert #4, and the remainder of the files will be copied.

The Programmer's Interface

The programmer's interface allows Sketch to be used as a tool by other programs. It is documented in t*he Programmer's Interface to Sketch* .

New Behavior for the Get Command

The action of the **Get** command was changed to be consistent with the TEdit **Get** command. It now deletes any sketch elements that are in the sketch prior to the **Get** command. The affect of the old **Get** command is available as the **Include** command on a submenu to the **Get** command.

Establishing Initial Defaults for Sketch

The variable SK.DEFAULT.FONT, if non-NIL, is used as the default font. If SK.DEFAULT.FONT is NIL, the default font (DEFAULTFONT) is used.

The following variables are used to establish the default setting for a new sketch. Descriptions of legal values can be found in the *Programmer's Interface to Sketch*. SK.DEFAULT.BRUSH is the default brush. SK.DEFAULT.ARROW.LENGTH is the default arrowhead size. SK.DEFAULT.ARROW.TYPE is the default type (one of LINE, CURVE, CLOSEDLINE or SOLID). SK.DEFAULT.ARROW.ANGLE is the default angle for arrowheads. SK.DEFAULT.TEXT.ALIGNMENT is the default text alignment. SK.DEFAULT.TEXTBOX.ALIGNMENT is the default textbox alignment. SK.DEFAULT.DASHING is the default dashing. SK.DEFAULT.TEXTURE, SK.DEFAULT.BACKCOLOR and SK.DEFAULT.OPERATION are combined to create the default filling.

1108 User's Guide Release Notes

What to Look For

The *1108 User's Guide* was extensively reorganized and rewritten for the Lyric Release. This made it nearly identical to the *1186 User's Guide*. This section contains a summary of changes affecting 1108 environments with the Medley release. Details are described in update pages available for the *1108 User's Guide.*

In every 1108 chapter that requires use of Lisp expressions of any kind, there is a notice regarding the use of **IL:** and a suggestion that expressions, functions, and variables be typed into an Interlisp Exec.

4. File System

Medley will accept floppy names up to 40 characters in length. Some of the Lyric font floppies have names in excess of 40 characters. Medley truncates the floppy name to 40 characters if asked to read a Lyric floppy with a longer name. The function FLOPPY.NAME is used to name a floppy. When it is not given any arguments, it returns the name stored on the floppy disk. When it is given a *NAME* argument, the floppy name is set to *NAME*. The 40 character limitation holds for both 1108 and 1186 floppies.

The {DSK} device on the 1108 and 1186 now accepts a wider range of characters in file names. Almost any character in character set 0 is acceptable. Previously, if you tried to create a file whose name included, for example, an underscore, you would see a "FILE NOT FOUND" error.

The 1108 and 1186 file systems had a problem with large partitions which would manifest itself as "HARD DISK error - can't find file page" when accessing newly created files. This would only appear on logical volumes larger than 64K pages. This problem has been fixed.

The function FILENAMEFROMID is now implemented.

6. System Tools

In System Tools, it is no longer necessary to execute a **Floppy Info!** command before attempting a **List!**.

The Medley System Tool now displays an error message when an NS Domain or Organization name is more than the allowed 20 characters long.

The Medley System Tool now supports sysout and microcode installation using the TCP FTP protocol. This feature may be used by selecting the "TCP/FTP" device type in the main System Tool window. Update pages for the *1108 User's Guide* describing this feature, are included with the Medley release.

7. Input/Output

Every time you allocate space on a floppy disk that has fewer than 200 free pages, a message is printed in the prompt window. That message gives an approximate number of free pages remaining after the allocation; it's intended to give you warning when your floppy is nearing full. The page count is correct only within +/- 2 pages because the message is printed in the course of the allocation, and the floppy's directory may grow when the new file is added to it .

8. Machine Diagnostics

Medley Boot Diagnostics for the 1108 include changed floppy disk names and slight changes in the prompts for running diagnostics from floppies.

1186 User's Guide Release Notes

What to Look For

The *1186 User's Guide* was extensively reorganized and rewritten for the Lyric Release. This section contains a summary of changes affecting 1186 environments with the Medley release. Details are described in update pages available for the *1186 User's Guide.*

In every 1186 chapter that requires use of Lisp expressions of any kind, there is a notice regarding the use of **IL:** and a suggestion that expressions, functions, and variables be typed into an Interlisp Exec.

1. Introduction

For Medley, the Xerox Lisp logo window has been changed to reflect the new name, Envos.

4. File System

Medley will accept floppy names up to 40 characters in length. Some of the Lyric font floppies have names in excess of 40 characters. Medley truncates the floppy name to 40 characters if asked to read a Lyric floppy with a longer name. The function FLOPPY.NAME is used to name a floppy. When it is not given any arguments, it returns the name stored on the floppy disk. When it is given a *NAME* argument, the floppy name is set to *NAME*. The 40 character limitation holds for both 1108 and 1186 floppies.

The {DSK} device on the 1108 and 1186 now accepts a wider range of characters in file names. Almost any character in character set 0 is acceptable. Previously, if you tried to create a file whose name included, for example, an underscore, you would see a "FILE NOT FOUND" error.

The 1108 and 1186 file systems had a problem with large partitions which would manifest itself as "HARD DISK error - can't find file page" when accessing newly created files. This would only appear on logical volumes larger than 64K pages. This problem has been fixed.

The function FILENAMEFROMID is now implemented.

5. Software Installation

The SKETCH executable files are too large to be contained on one floppy. The files are now distributed on two floppies: *Medley Library Floppy #3* and *Medley Library Floppy #4*. To load SKETCH, type the Interlisp exec command:

(FILESLOAD LOAD-SKETCH)

The LOAD-SKETCH function will copy all SKETCH files from *#3*; then prompt you to insert *#4*, and the remainder of the files will be copied.

6. System Tools

In System Tools, it is no longer necessary to execute a **Floppy Info!** command before attempting a **List!**.

The Medley System Tool now displays an error message when an NS Domain or Organization name is more than the allowed 20 characters long.

The Medley System Tool now supports sysout and microcode installation using the TCP FTP protocol. This feature may be used by selecting the "TCP/FTP" device type in the main System Tool window. Update pages for the *1186 User's Guide* describing this feature, are included with the Medley release.

7. Input/Output

Every time you allocate space on a floppy disk that has fewer than 200 free pages, a message is printed in the prompt window. That message gives an approximate number of free pages remaining after the allocation; it's intended to give you warning when your floppy is nearing full. The page count is correct only within +/- 2 pages because the message is printed in the course of the allocation, and the floppy's directory may grow when the new file is added to it.

The following function applies only to 1186 users:

(**dove.xor.cursor** *&optional xor-p)* [Function]

If no argument is given, this function returns the current state of the 1186 cursor (nil implies an or'ing cursor, t an xor'ing cursor). If an argument is given, changes the state of the 1186 cursor appropriately.

8. Diagnostics

Medley Boot Diagnostics for the 1186 include changed floppy disk names and slight changes in the prompts for running diagnostics from floppies.

[This page intentionally left blank]

7. IMPLEMENTATION

This section describes new features and enhancements that implement Common Lisp into the Lisp operating environment within the Medley release. This information supplements the *Common Lisp Implementation Notes,* Lyric release. Medley enhancements are indicated with revision bars in the right margin.

New Features Since Lyric

The following description summarizes the new Common Lisp implementation features that have been added or changed since the Lyric release.

New compiler Interface -- The Medley compiler gives better progress reports and it is now possible to invoke the compiler on any definer (not just functions, as before).

New Implementation of Defstruct -- A new version of defstruct compiles more compactly and gives more options so that defstruct has at least as much functionality as the Interlisp record package.

Adoption of features and clarifications suggested by the Common Lisp Cleanup Committee -- Among other changes, the behavior of **append** on dotted lists is now better defined, and a new function **xcl:row-major-aref** has been added.

Common Lisp Veneer on the Interlisp record package -- A collection of macros that make the use of existing Interlisp datatypes more appealing has been added.

Performance enhancements -- A closure caching scheme now insures that repeated calls to symbol-functions of the same symbol will return EQ compiled-function objects.

New opcodes have been added for several common list functions, such as **member** and **assoc**.

Common Lisp Definers

The Medley release contains a new implementation of definers and a reworking of the top level of the XCL Compiler. These represent upward compatible changes that have the effect of allowing the Common Lisp compiler to print out progress reports indicating which definer is currently being compiled. To receive the full benefit of these changes, recompile any file containing a **defdefiner** expression.

It is now possible to compile individual definers by using any of the following forms:

Compile-Definer

(**xcl:compile-definer** *name type*)

Compile and install the definer of type *type* named *name* .

EXAMPLE:

```
(xcl:compile-definer 'foo 'structures)
```

In this example, the definer will compile and install the structures definition of foo.

Compile-Form

(**xcl:compile-form** *form*)

Compile and evaluate *form*.

EXAMPLE:

```
(xcl:compile-form '(progn (defconstant c 1) (defun foo (a b) (+ c a
b))))
```

In this example, the definer will compile and evaluate the progn using compile-file semantics.

EXAMPLE:

```
(xcl:compile-form '(with-collection (dotimes (i 10) (collect i))))
```

In this example, the definer returns:

(0 1 2 3 4 5 6 7 8 9)

Define-File-Environment

Rather than establishing **il:makefile-environment** props and **il:filetypes** on the root name of a file, you can define a file environment using the form:

(**xcl:define-file-environment** *filename* &key *readtable package base compiler*)

This produces an object of file-manager type **xcl:file-environment**s. The *filename* can be either a string or a symbol. The rootname of the file is constructed by interning the *filename* in the Interlisp package. Puts the *compiler* argument (if any) under the **il:filetype** prop of the file rootname. Puts the *readtable, package* and *base* arguments (if any) under the **il:makefile-environment** prop of the file rootname. None of the arguments are evaluated. There are no defaults.

EXAMPLE:

```
(xcl:define-file-environment myfile :package "XCL-USER" :readtable "XCL"
:compiler :compile-file)
```

In this example, *compile-file* is put under the **il:filetype** prop of *myfile.* The *readtable*, *XCL* and *compile* arguments are put under the **il:makefile-environment** prop of *myfile.*

NOTE: **xcl:define-file-environment** is a definer and hence will not be installed if **il:dfnflg** is **il:prop** or if a file is prop loaded.

Site-Name Special Uses

The following special variables are defined and may be set in your init file to inform Common Lisp of site information:

xcl:*short-site-name*

This variable is used in the function **short-site-name.**

xcl:*long-site-name*

This variable is used in the function **long-site-name**.

EXAMPLES:

```
(setq xcl:*short-site-name* "AIS")
(setq xcl:*long-site-name* "Artificial Intelligence Systems")
```

In these examples, (short-site-name) returns "AIS" and (long-site-name) returns "Artificial Intelligence Systems".

Record Access

The Medley release contains several methods for accessing existing Interlisp records using Common Lisp syntax. These features help to integrate Interlisp and Common Lisp. The following sections describe these additions.

Define-Record

(**xcl:define-record** *name interlisp-record-name*
&key *conc-name constructor predicate fast-accessors*) [Definer]

Creates a structures object named by the symbol *name* that provides Common Lisp accessors, settors, predicates and constructors for the Interlisp record named by the symbol *interlisp-record-name*. The Interlisp record must be defined before the **xcl:define-record** expression is evaluated. The keyword arguments are treated as in defstruct. The package of constructed names is taken from the value of ***package*** at the time of evaluation (as in defstruct). The system contains no predeclared **define-record**s.

EXAMPLE:

The `form:`

```
(xcl:define-record menu il:menu)
```

allows you to write:

```
(menu-items foo)  and (setf (menu-items foo) fie)
```

rather than:

```
(il:fetch (il:menu il:items) il:of foo)
```

Record-Fetch

(**xcl:record-fetch** *record field object*) [Macro]

Evaluates *object*. Does not evaluate *record* and *field.* Both *record* and *field* must be symbols. Symbols with the same p-names are interned in the Interlisp package and are used to construct an **il:fetch** form. **xcl:record-fetch** may be used with **setf** and expands to the suitable replace form.

Record-FFetch

(**xcl:record-ffetch** *record field object*) [Macro]

Similar to **xcl:record-fetch**, but an **il:ffetch** form is generated instead. Evaluates *object.* Does not evaluate *record* and *field* . Both *record* and *field* must be symbols. Symbols with the same p-names are interned in the Interlisp package and are used to construct an **il:ffetch** form. Ffetch may be used with **setf** and expands to the suitable freplace form.

Record-Create

(**xcl:record-create** *record* &rest *keyword-pairs*) [Macro]

Evaluates the second element of each pair. Does not evaluate *record (record* must be a symbol) . A symbol with the same p-name is interned in the Interlisp package and used to construct an **il:create** form. The rest of the arguments form keyword pairs. The first element of each pair should be a symbol such that a symbol with the same p-name exists in the Interlisp package and names either a valid slot for this record or is one of **:using, :copying, :reusing**, or **:smashing**.

Array Reference

(**xcl:row-major-aref** *array index*) [Function]

Returns the element of *array* given by the row-major-index *index*. The array can be of any dimension. This function can be used with **setf** .

Shadowing of Global Macros

The XCL Compiler now properly handles shadowing of global macros by lexical functions. In the Lyric Compiler, lexical functions defined with **flet** did not shadow global definitions of the same name. This has been fixed in Medley.

Evaluating Load-time Expressions

The XCL Compiler now handles **il:loadtimeconstant** correctly. The new Compiler substitutes the entire expression for each reference to the value of a load-time constant. There are potential problems if the code depends on the expression being evaluated exactly once, e.g. if it contains (IDATE).

Common Lisp Defstruct Options

The Medley release contains a new implementation of **defstruct** that offers greater compiled-code compaction, and several new extensions that increase efficiency. This implementation introduces functionality that allows **defstruct** to parallel the Interlisp

record module in flexibility. These features also help to integrate Interlisp and Common Lisp. The following sections describe these additions.

Defstruct Options

:inline

Can be one or both of :accessor and :predicate or t, implying '(:accessor :predicate) or nil, implying no optimizations allowed or :only, implying all accessors and the predicate will be inline only and not funcallable (not usable with the Lisp primitive "**funcall**"). The default is '(:accessor :predicate).

Copiers and constructors are never inline. The option (:inline :only) implies that no funcallable accessors will be generated (similarly, the predicate, if any, will not be funcallable).

:fast-accessors

Can be t or nil. t implies inline accessors will not type check. The default is nil.

Note that funcallable accessors (if any), always type check, if possible.

NOTE: This represents a change from the Lyric implementation, which allowed specification of a list of slot names that had fast inline accessors.

:template

Can be t or nil, t implies that no datatype will be instantiated. (:template t) implies no :type option. The default is nil.

Templated defstructs have no predicates, copiers or constructs. It is an error to supply any such option in combination with (:template t). Templated defstructs are intended to be used as are IL:blockrecord's. It is possible for a templated defstruct to include another templated structure, but it is an error for a standard defstruct to include a templated structure.

Funcallable accessors (accessors that may be used with the Lisp primative "funcall") share code with suitable closure templates if the defstruct is compiled with the XCL Compiler. Byte compiled defstructs still generate explicit defun's for all funcallable accessors.

Defstruct Slot Options

:type

The following specialized types are recognized:

(unsigned-byte {1 - 16})

(signed-byte {16, 32})

float, etc.

(member t nil)

il:fullpointer

il:xpointer

il:fullxpointer

Warning When Using Defstruct

Defstruct automatically generates a number of auxilliary functions without checking whether redefining those functions will affect the system. To avoid redefining key functions, you should be aware of the names that will be used. For example:

Do not attempt to define a Structure named TREE. This use of Defstruct implicitly redefines the built-in Common Lisp function COPY-TREE, which renders your system inoperable.

If you have already tried to define a (DEFSTRUCT TREE A B) structure by mistake, you will need to reload your system.

Macros for Collecting Objects

xcl:with -collection

(**xcl:with-collection &body** *forms*) [Macro]

(**xcl:collect** *form*) [Macro]

This pair of macros is provided for efficiently collecting objects into a list. In Common Lisp, there is no direct facility provided for doing this, so one must either push objects onto a list, then reverse it, or maintain a tail pointer to the list and use **rplacd** to add new items. The latter has an efficient implementation in Xerox Common Lisp, and **xcl:with-collection** is provided to take advantage of it.

Lexically within the body of an **xcl:with-collection**, the macro **xcl:collect** is defined. It will append the value of its argument to the end of the list being collected. The value of **xcl:with-collection** is the collected list.

xcl:collect may be used inside of functions passed as arguments to other functions.

EXAMPLE:

```
(xcl:with-collection
  (maphash
    #'(lambda (key val)
        (when (interesting-p val) (xcl:collect key)))
    the-hash-table))
```

will collect a list of all the "interesting" keys in the order that they were encountered.

It is an error to use **xcl:collect** outside the scope of an **xcl:with-collection.** Proper lexical nesting is observed, so an instance of **xcl:collect** applies to the most deeply nested **xcl:with-collection** that is is found in.

Macros for Writing Macros

xcl:once-only

(**xcl:once-only** *({ variable }*)* &body *forms*) [Macro]

This macro is provided to aid in writing macros. **xcl:once-only** helps solve the problem of multiple evaluation of subforms of a macro.

EXAMPLE:

```
(defmacro test (reference form)
  `(setf ,reference (cons ,form ,form)))
```

This example has the problem that **form** will be evaluated twice. To avoid this, one might instead write:

```
(defmacro test (reference form)
  (let ((value (gensym)))
    `(let ((,value ,form))
       (setf ,reference (cons ,value ,value)))))
```

This solves the problem of multiple evaluation, but introduces some others. If **form** is in fact something simple, like a reference to a variable or a literal, there was no need to create the temporary variable, thus "wasting" a symbol. This can be extremely important in Xerox Common Lisp as symbol space is limited and symbols are never reclaimed. If there are many temporary values to be computed, the macro definition becomes cluttered with calls to gensym that obscure the essence of the code.

xcl:once-only helps solve these problems. For each of the variables listed, **xcl:once-only** determines if its value (at macroexpansion time) is simple: a symbol or a literal. If it is, appearances of that variable in the macroexpansion will remain unchanged. If it is not, the macroexpansion will contain code to store the value in a temporary **gensym'**ed variable and use that variable in the macroexpansion. Thus, the example could be written as

```
(defmacro test (reference form)
  (xcl:once-only (form)
    `(setf ,reference (cons ,form ,form))))
```

Then `(test (aref the-array x) y)` will expand to something like

```
(setf (aref the-array x) (cons y y))
```

while `(test (aref the-array x) (random-form))` will expand to something like

```
(let ((#:g377 (random-form)))
  (setf (aref the-array x) (cons #:g377 #:g377)))
```

Note that **xcl:once-only** does not attempt to preserve order of evaluation. If this is important then you will still have to create temporary variables yourself.

Common Lisp Append Datatypes

A clarification adopted by X3J13 involves the behavior of the APPEND function with non-lists. The cdr of the last cons in any but the last argument given to APPEND is discarded (whether NIL or not) when preparing the list to be returned. In the case where there is no last cons (i.e., the argument is not a list) in any but the last list argument, the entire argument is effectively ignored. In this situation, if the last argument is a non-list, the result of APPEND can be a non-list. NB: APPEND and COPY-LIST now produce different results for non-lists.

EXAMPLE:

```
(append '(a b c . d) '())
```

produces the result:

```
(a b c)
```

EXAMPLE:

```
(append '(a b . c) '() 3)
```

produces the result:

(a b . 3)

EXAMPLE:

```
(append  3  17)
```

produces the result:

17.

Closure Cache

The Medley sysout contains a closure cache that provides increased time and space efficiency. Less new memory is allocated because repeated calls to symbol-function of the same symbol now will cons exactly one closure object. Repeated calls to symbol-function of the same symbol now return EQ- compiled function objects.

Symbols and Packages

Pkg -goto and In-package

PKG-GOTO is now a synonym for IN-PACKAGE. The PKG-GOTO function can be used to change packages in an exec.

PKG-GOTO takes one argument, which can be either a double-quoted string, a symbol, or a package structure. This function is used to set package in an exec.

(**xcl:pkg-goto** *package-name* &key *nicknames use*) [Function]

PKG-GOTO operates like IN-PACKAGE, but asks for confirmation if a new package is being created. The function is useful at the top level in the exec, to avoid creating new packages when a name is misspelled.

Defpackage Export argument

Defpackage's EXPORT argument now accepts strings. Optionally, strings can be given to :EXPORT instead of symbols. This is recommended when defpackage is used in the makefile-environment property of a file. The strings are interned in the package being defined and then exported.

Debugging Tools

Breaking

Even with HELPDEPTH set to zero, some errors do not cause a break. In Koto and the old Interlisp execs in Lyric, the workaround is:

(SETTOPVAL 'HELPFLAG 'BREAK!)

In Medley and Lyric's new execs, HELPFLAG is bound but not continually reset. The workaround:

(SETQ HELPFLAG 'BREAK!)

affects the current exec until the next time you call RESET (or control-D). If you want the change in HELPFLAG to be seen by other processes, you still need to use SETTOPVAL, and RESET any execs in which you want to see the effect.

For related information, see the Medley error system variable XCL:*BREAK-ON-SIGNALS* described in Appendix E.

Advising

In Lyric, putting a second piece of advice on a function caused the system to believe that the function was in fact not advised, so any further advice threw out the already existing advice. This has been fixed. In Medley, the correct list entries are made regardless of whether the function was previously advised.

In Lyric, loading a file with advice caused multiple instances of the advice to be instantiated. To prevent this, ADVISE is now changed in Medley in the following way: When a new piece of advice is put on a function, the system examines the already existing advice to see if the some advice is already there. If so, the old advice is removed before adding the new advice. Sameness is determined by a test similar to CL:EQUALP, except that *case* distinctions are significant in strings and characters. The priority and location of the advice is taken into account when determining the "sameness." This makes it possible, for instance, to have identical advice be both :FIRST and :LAST.

Advice is no longer replicated when loaded more than once.

The debugger and inspector now display interpreted lexical closures conveniently. Displayed lexical closure contents include the function contained, and any lexical bindings in the closure. Compiled closures are not conveniently inspectable. Common Lisp eval stack frames show their associated lexical environment in a similar manner.

The :when option to XCL:BREAK-FUNCTION no longer causes the broken function to return NIL when the break is not taken. The correct values are returned.

Argument Names Displayed for Interpreted Functions

In the debugger, the frame inspector window will now display the argument names for interpreted Common Lisp functions. Previously, it gave them pseudonames "arg0" "arg1" etc.

Lexical Variables Evaluated by Debugger

The debugger EVAL command now evaluate expressions in the lexical environment --i.e., you can evaluate an expression and use variables that are lexically bound in your code. Only the lexical environment at the point of the break can be evaluated. You can't presently back up to any given lexical environment.

EXAMPLE:

```
(defun fact(x)(if(= 1 x)nil(*x(fact(1-x)))))
(fact 4)
```

;; breaks. if you then type

```
EVAL x
2
```

Pathname Component Fixed in FS-ERROR

In Lyric, only one of the three FS-ERROR conditions was passed a pathname component, resulting in the File Cacher not knowing which file had the error, or resulting in pathname being lost when PROTECTION VIOLATION or FILE SYSTEM RESOURCES EXCEEDED were signaled. This problem occurred most noticeably in Lyric when Interlisp errors were translated to XCL. This condition has been fixed in Medley. FS-ERROR now correctly receives all the pathname components.

Compiler Optimizations

Warning when using LABELS construct

In Lyric, use of the LABELS construct generated circular structure that would not get collected. Interpreted, a LABELS construct always creates this non-collectible structure. Compiled, such structure would be created if there were non-tail-recursive or mutually referencing subfunctions. The values of any closed-over variables are captured by this structure and thus also not collected, potentially causing large storage leaks. The latter situation has been relieved somewhat for Medley.

In Medley, the unavoidable circularity has been reduced to include only the mutually referencing functions, but not any of the other data that they access. Thus, the uncollectable structure is created only when a new copy of the code blocks are created, such a by compiling the function containing the LABELS rather than each time that function is called.

COMS added to dfasl files

The Medley compiler has been modified to better handle the il:define-file-info, and defpackage forms. Now, loading a dfasl file

is not implicitly SYSLOAD. Since the file COMS for the file is now included in the dfasl, that file will be noticed by the file manager unless the load is explicitly SYSLOAD. (SYSLOADing of compiled lcom and dfasl files is recommended.)

In Lyric, dfasls of file manager files did not contain the COMS of the file. In Medley, COMS are present in dfasl files, just as they are in lcom files. As with lcom files, the COMS will not be loaded when the LDFLG argument to LOAD is SYSLOAD, nor will the name of the file be added to FILELST, but instead will be added to SYSFILES.

Note: We discourage loading either sort of compiled file (lcom or dfasl) with any value for LDFLG but SYSLOAD. Unless you intend to edit a file, you should always load it SYSLOAD. Even when you intend to edit it, it is usually preferable to SYSLOAD it and then load the source PROP. If there are too many source files for this to be practical, we recommend use of the WHERE-IS Library module.

While the location of definitions is made known to the edit interface when files are loaded, it can be very inefficient when files are not SYSLOADed. If, for example, you load ten compiled files with LDFLG=NIL and then evaluate (ED 'FOO), then the COMS of all ten files must be searched for definitions of each manager type with name FOO. With forty manager types this comes to 400 parses of COMS -- a time-consuming operation. If you instead load the compiled files SYSLOAD and the sources PROP, then no COMS need be searched, as checking for definitions of each manager type is sufficient.

Loadflg argument

The Medley release contains a new keyword argument to **cl:load.**

(**cl:load** *filename* &key *verbose print if-does-not-exist loadflg***)**

The *loadflg* argument follows the sematics of the loadflg argument to **il:load**, with the exception that the loadflg argument will always be interned in the Interlisp package.

EXAMPLE:

```
(cl:load "Mycompiled-file.dfasl" :loadflg :sysload)
```

In this example, "Mycompiled-file.dfasl" will load without the file manager noticing that file.

Note: As explained in the previous section, we discourage loading either sort of compiled file (lcom or dfasl) with any value for *ldflg* but SYSLOAD.

Changes in CL:MAP, CL:WRITE-STRING, CL:COERCE , CL:GENSYM and IL:DEFERREDCONSTANT

In Lyric, a compiled call to CL:MAP that had been used for effect would occasionally cons up a new list anyway. It would fail in the case that the first argument was a constant that evaluated to NIL, but not NIL itself, e.g. 'NIL. This has been fixed and no longer occurs in Medley.

CL:WRITE-STRING is now twice as fast and creates no new structure.

CL:COERCE now correctly returns the original object in all cases where Common Lisp and Lisp require it.

The CL Compiler now compiles CL:GENSYM properly.

IL:DEFERREDCONSTANT is now handled correctly by the XCL compiler.

ADD.PROCESS no longer coerces the process name to a symbol. Rather, process names are treated as case-insensitive strings. Thus, you can use strings for process names, and when typing process commands to an exec, you need not worry about getting the alphabetic case correct.

Compiler keeps Special &REST arguments

The CL Compiler now retains special &REST arguments. The Lyric compiler threw away special &REST arguments. This has been fixed in the Medley CL Compiler.

Compiler ignores TEdit formatting

COMPILE-FILE will now ignore TEdit formatting, but only if TEdit is loaded.

Compiler notices Tail-recursive Lexical Functions

The XCL Compiler now performs tail recursion elimination on FLETed lexical functions.

Compiler Error Message "BUG: Inconsistent stack depths seen"

You may occasionally see this error message while compiling. Normally, error messages from the compiler beginning with "BUG" indicate an internal compiler error. In this particular case, the compiler error may reflect an error in the code you are compiling.

There is currently no compile-time argument checking. The compiler performs an optimization that turns a tail-recursive function call into a jump back to the beginning of the function. If this tail-recursive call has the wrong number of arguments, the stack modeler in the assembler will detect this as incosistent stack depths, leading to the above error message.

EXAMPLE:

```
(defun bad-length (x n)
  (if (endp x) n (bad-length (cdr x))))
```

Compiling this form will result in the error "BUG: Inconsistent stack depths seen." The recursive call to bad-length has only one argument, but the function expects two.

Thus, if you see this error message, you should check for tail-recursive function calls with the wrong number of arguments.

Format ~C and WRITE-CHAR

In accordance with a recommendation of X3J13, the ~C FORMAT operation with no modifiers now behaves exactly the same as WRITE-CHAR for characters with no bits. The Medley release of XCL conforms to this; the Lyric release did not. If you need to obtain the Lyric behavior of ~C, use ~:C.

WITH-OUTPUT-TO-STRING and WITH-INPUT-FROM-STRING

For consistency with WITH-OPEN-STREAM and WITH-OPEN-FILE, WITH-OUTPUT-TO-STRING and WITH-INPUT-FROM-STRING now close the stream on exit from the form. WITH-OUTPUT-TO-STRING is now significantly faster when writing long strings.

[This page intentionally left blank]

Replace this page with Table of Contents tab

Replace this page with Preface tab

Replace this page with
1. Introduction
tab

Replace this page with 2. Notes and Cautions tab

Replace this page with
3. Integration of Languages tab

Replace this page with 4. Changes to Interlisp-D tab

Replace this page with 5. Library Modules tab

Replace this page with
6. User's Guides
tab

Replace this page with 7. Common Lisp Implementation tab

Replace this page with
A. The Exec
tab

Replace this page with
B. SEdit
tab

Replace this page with
C. ICONW
tab

Replace this page with D. Free Menu tab

Replace this page with
E. Error System
tab

Replace this page with Index tab

TAB TEXT FOR RELEASE NOTES MANUAL

POINT SIZE:

FONT: MODERN

Table of Contents	Preface	1. Introduction	2. Notes and Cautions	3. Integration of Languages
4. Changes to Interlisp-D	5. Library Modules	6. User's Guides	7. Common Lisp Implementation	A. The Exec
B. SEdit	C. ICONW	D. Free Menu	E. Error System	Index

TABS FOR MEDLEY RELEASE NOTES MANUAL (2-1/4" BINDER)

TYPE: MAJOR
TAB SIZE: 2-3/8"
NO. TABS PER BANK: 5
NO. OF BANKS: 3

Point size: 10
Font : Modern

COLOR OF TABS: GREEN background, BLACK lettering

Point size: 10
Font : Modern

x Tick marks indicate hole placement:
3-hole punch, 5/16" diameter holes

1st hole centered: 1 3/16" from top of page, 7/16" in from side
2nd hole centered: 5 7/16" from top of page, 7/16" in from side
3rd hole centered: 9 11/16" from top of page, 7/16" in from side

BANK 1

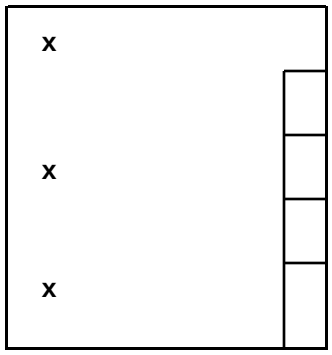
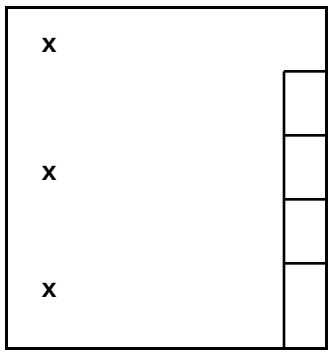


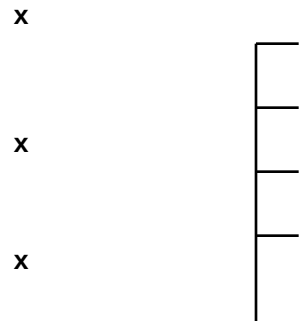
Table of Contents
Preface
1. Introduction
2. Notes and Cautions
3. Integration of Languages

BANK 2



4. Changes to Interlisp-D
5. Library Modules
6. User's Guides
7. Common Lisp Implementation
A. The Exec

BANK 3



B. SEdit
C. ICONW
D. Free Menu
E. Error System
Index

TAB TEXT FOR MEDLEY RELEASE NOTES MANUAL

POINT SIZE: 12

FONT: MODERN

Table of Contents	**Preface**	**1. Introduction**	**2. Notes and Cautions**	**3. Integration of Languages**

4. Changes to Interlisp-D	**5. Library Modules**	**6. User's Guides**	**7. Common Lisp Implementation**	**A. The Exec**

B. SEdit	**C. ICONW**	**D. Free Menu**	**E. Error System**	**Index**

TAB SPECIFICATIONS for

LISP RELEASE NOTES, MEDLEY RELEASE
and
LISP LIBRARY MODULES, MEDLEY RELEASE

TABS

Tab Paper Size 8 1/2 by 11 inches (excluding the actual tab)

Stock Type Ledger; matte finish

Stock Weight 110 pound

Stock Color White

Tab Coating Mylar

Mylar Color
PMS 422-C coordinated gray

Holes 3-hole punch, 5/16" diameter holes

Hole Reinforcement clear mylar strip

Hole Placement ***1st hole centered:*** 1 3/16" from top of page, 7/16" in from side
2nd hole centered: 5 7/16" from top of page, 7/16" in from side
3rd hole centered: 9 11/16" from top of page, 7/16" in from side

Tab Cut Requirements

Number of Tab Cuts 5

Size of Cuts (Length) 2 3/8"

Depth of Tab Cut 1/2 inch

Cut Shapes straight edge, curved top, curved bottom

Print Specifications

One or Both Sides Both sides

Font Modern/Optima

Point Size 10

Ink Color Black

Type Spacing Centered

PACKAGING INSTRUCTIONS: Shrink wrap with text

TOTAL NUMBER COPIES: 500

LISP RELEASE NOTES

400006
Medley Release 1.0
September 1988

Address comments to:
ENVOS
User Documentation
1157 San Antonio Rd.
Mountain View, CA 94043
415-966-6200

LISP RELEASE NOTES

Medley Release 1.0

400006

September 1988

Text was written and produced with Envos formatting tools using Xerox printers to produce text masters. The typeface is Modern.

TABLE OF CONTENTS

[This page intentionally left blank]

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 `/ur/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 inthe 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.

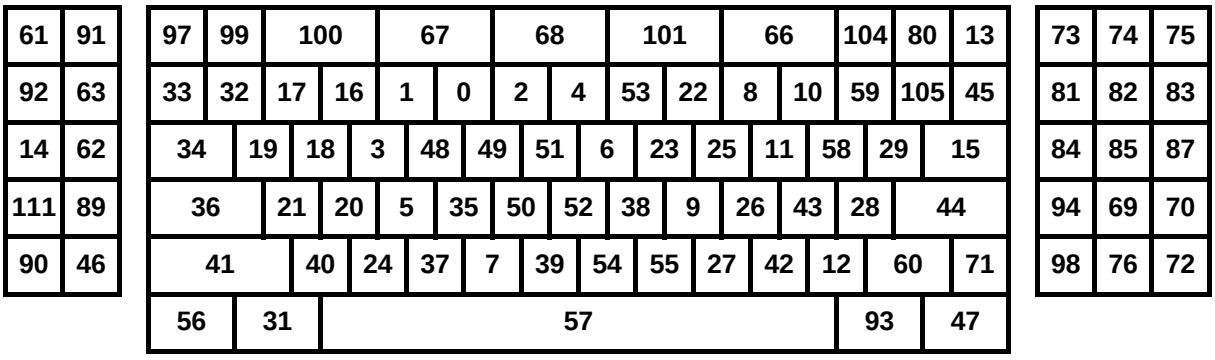


Figure 3-1. Sun Type 3 Key Numbering

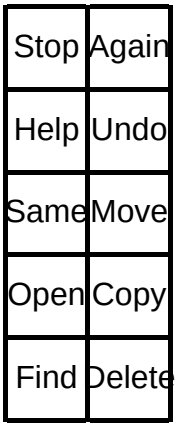


Figure 3-2. Sun Type 3 Left Key Pad

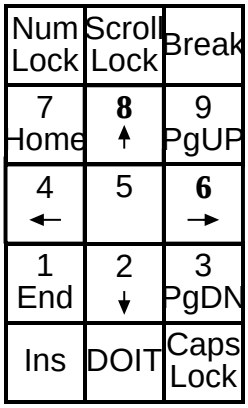


Figure 3-3. Sun Type 3 Right Key Pad

Center	Bold	Italic		Case		Strikeout		Underline		Super Sub		Larger Smaller	Margin	Back Word
Esc	! 1	@ 2	# 3	$ 4	% 5	^ 6	& 7	* 8	(9	) 0	_ -	+ =	| \	~ ‘
Tab	Q q	W w	E e	R r	T t	Y y	U u	I i	O o	P p	{ [	}]	Backspace	
Ctrl	A a	S s	D d	F f	G g	H h	J j	K k	L l	: ;	" ’	Return		
Shift	Z z	X x	C c	V v	B b	N n	M m	< ,	> .	? /	Shift	LF		
Caps	Meta	Space										Expand	Next	

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
```

61	91		97	99	100	67	68	101	66	104	80	106	107	108	105	13		75	110	74	73					
109	63		33	32	17	16	1	0	2	4	53	22	8	10	59	15		64	65	95	96					
14	89		34	19	18	3	48	49	51	6	23	25	11	58	29	0	44	81	82	83	102					
111	62		36		21	20	5	35	50	52	38	9	26	43	28	45		84	85	87						
90	46		41			40	24	37	7	39	54	55	27	42	12	60	71	94	69	70	76					
92			56	31	86	57									88	93	47	98		13						

Figure 3-5. Sun Type 4 Key Numbering

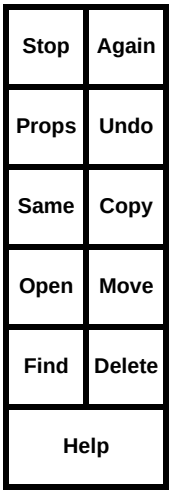


Figure 3-6. Sun Type 4 Left Key Pad

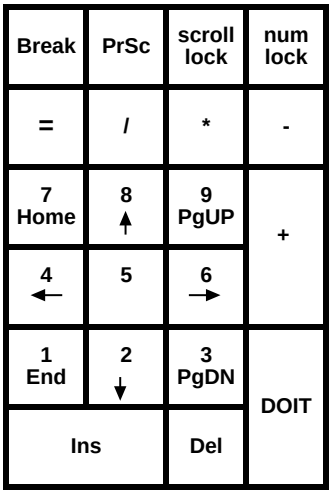


Figure 3-7. Sun Type 4 Right Key Pad

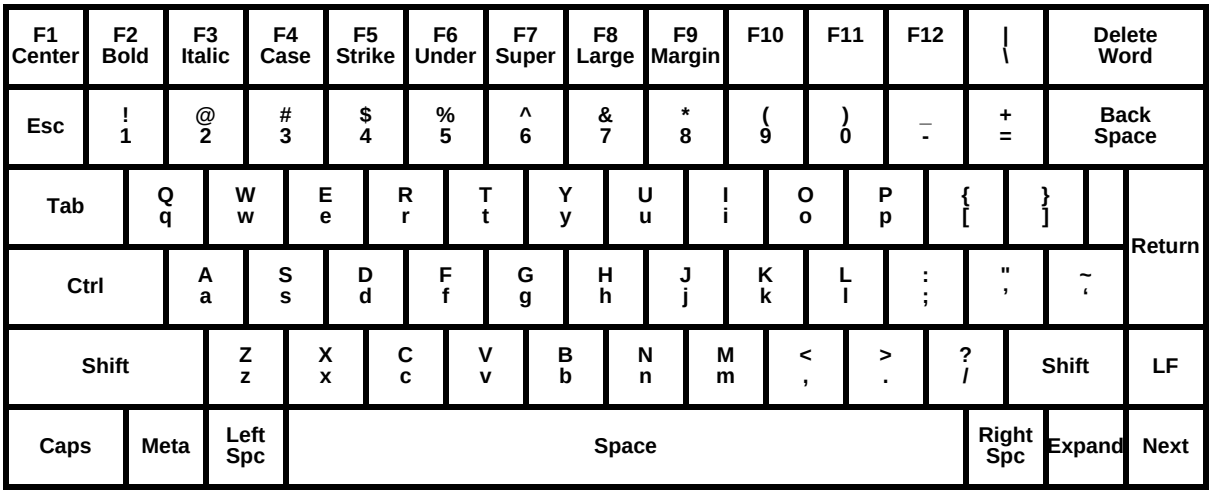


Figure 3-8. Sun Type 4 Center Key Pad Interpretation

[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 re 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 hexadecimal 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 thetape 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.
- *Medleyfor 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

R

S

T