CHAPTER 18

INTERLISP-D SPECIFICS

Interlisp-D is an implementation of the Interlisp language that runs on the Xerox 1100, 1108, and 1132 machines. It is completely upward compatable with the older Interlisp-10, except as specified in this manual. The most significant extension to Interlisp is the window display package, described on page 19.1. However, Interlisp-D also offers many other extensions, which are described in detail below.

18.1 INTERLISP-D INTERRUPT CHARACTERS

The table below gives the interrupt characters currently enabled in Interlisp-D. Many of these are the same as those used in the Tenex version of Interlisp-10, but some have been removed, and some have had their meanings changed. It is possible to change the assignments of control characters to interrupts using `INTERRUPTCHAR` (page 9.17).

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. Most of the interrupt characters below refer to the TTY process, which is the one currently receiving keyboard input. Control-H can be used to break arbitrary processes. For more information, see page 18.35.

control-B — Causes a break within the TTY process. Use control-H to break a particular process. Note that this break occurs at the next function call, so it is like control-H in Interlisp-10; it is always safe to resume the computation. There is no interrupt character like control-B in Interlisp-10

control-C — On the Xerox 1100 and Xerox 1132, brings the user into the Raid low-level debugger. From Raid, typing control-N resumes the Lisp computation, and control-D resets the stack. On the Xerox 1108, after typing control-C, the system stops and waits for the next character typed. Pressing the `STOP` key will do a `HARDRESET`, returning control to the user. Pressing the `UNDO` key will start up the TeleRaid debugger.

control-D — Aborts the TTY process, and unwinds its stack to the top level. Calls `RESET` (page 9.14).

control-E — Aborts the TTY process, and unwinds its stack to the last `ERRORSET`. Calls `ERROR!` (page 9.14).

control-H — Pops up a menu listing all of the currently-running processes. Selecting one of the processes will cause the break to take place in that process.

control-P — Changes the `PRINTLEVEL` setting, as described on page 6.18.

control-T — Prints status information for the TTY process.

Note: The control-O, and control-S interrupt characters from the Tenex version of Interlisp-10 are not enabled in Interlisp-D.

18.2 GARBAGE COLLECTION

Interlisp-D has a reference-counting garbage collector (Interlisp-10 uses the more familiar mark-and-sweep algorithm). A reference-counting garbage collector uses time proportional to the garbage being collected and not to the size of the address space. This is a crucial advantage for a large address space system such as Interlisp-D. It does have a disadvantage in that circular lists are never reclaimed, as their reference count never goes to zero. In addition, atoms are currently not garbage collected; and non-atomic hash array keys are not collected (in Interlisp-10, when a non-atomic hash key is no longer referenced except by the hash array itself, the hashlink goes away and both the key and the value, if it is nowhere else referenced, are reclaimed).

Garbage collection in Interlisp-D is controlled by the following functions and variables:

`(RECLAIM)` [Function]

Initiates a garbage collection. `RECLAIM` always returns 0, independent of the actual number of cells collected.

`(RECLAIMMIN N)` [Function]

The frequency of garbage collection is user settable via the function `RECLAIMMIN` (which plays a role similar to Interlisp-10's `MINFS`, which is a no-op in Interlisp-D). Lisp keeps track of the number of cells of any type that have been allocated; when it reaches the `RECLAIMMIN` number, a garbage collection occurs. `(RECLAIMMIN N)` returns the current setting of the parameter, and, if `N` is non-`NIL`, sets it to `N`.

As there is no motivation for the Interlisp-10 `CTRL-S` interrupt, it is not enabled.

`RECLAIMWAIT` [Variable]

Interlisp-D will invoke a `RECLAIM` if the system is idle and waiting for user input for `RECLAIMWAIT` seconds (currently set for 4 seconds).

`(GCGAG MESSAGE)` [Function]

`GCGAG` sets the message that appears on the display screen while a garbage collection is taking place. If `MESSAGE` is non-`NIL`, the cursor is complemented during a `RECLAIM`; if `MESSAGE` = `NIL`, nothing happens. This limited choice exists because it was found that printing a message took a significant fraction of the time of small `RECLAIM`s. The value of `GCGAG` is its previous setting.

`(GCTRP)` [Function]

The function `GCTRP` returns the number of cells (of any type, not just `LISTP`) until the next garbage collection, according to the `RECLAIMMIN` number, although this number is not very meaningful.

18.3 VARIABLE BINDINGS

Interlisp-D uses deep binding of variables, whereas Interlisp-10 currently uses shallow binding (prior to 1975, Interlisp-10 used deep binding). Although this makes little difference for most programs, it can make a difference in efficiency of execution. For example, it is better to pass parameters as arguments than to let subfunctions reference them freely. In addition, declaring variables that are never bound (i.e., whose top level value only is used) to be GLOBALVARS is important. Sloppy Interlisp-10 code that rebinds variables that have been declared as GLOBALVARS will not run correctly in Interlisp-D. Be careful to use RESETVARS to ''rebind'' variables that are declared GLOBALVARS. RESETVARS works in both systems; in a shallow system, RESETVARS just binds its arguments as PROG variables (and makes sure they are declared SPECVARS), while in a deep system such as Interlisp-D, entries are made on RESETVARSLST. If the compiler sees an attempt to bind a global variable, it will print out an error message.

For performance reasons, it is important to declare global variables as such in Interlisp-D. This can be done with the GLOBALVARS file package command (page 11.25), which causes variables to be declared as global to the compiler. For more information on variable bindings and performance, see page 18.19.

18.4 STACK FORMAT

Both the interpreter and compiler generate different intermediate frames than are found in Interlisp-10, so if the user has code that assumes a particular number of frames will exist at some point (e.g., using STKNTH), it will probably be wrong. STKPOS and STKSCAN are still available, however, and REALSTKNTH and REALFRAMEP are useful for ignoring those intermediate frames.

18.5 SAVING VIRTUAL MEMORY STATE

The Interlisp-D virtual memory is kept in the file Lisp.virtualmem. As virtual memory pages are accessed, they are loaded from this file into real memory. To exit from Interlisp-D to the Alto Executive so that it is possible to return to the current Interlisp-D environment, it is necessary to save the state of the virtual memory. The simplest way is to use the function LOGOUT (page 14.2). This will write out all altered pages from real memory to Lisp.virtualmem.

If you are the sole user of Interlisp-D on a disk partition, then you will probably want to use LOGOUT. However, if other Interlisp-D users may be using that partition, and you wish to save your state, then it may be more appropriate to use SYSOUT (page 14.3). Note that SYSOUT in Interlisp-D saves the *entire* state of the virtual memory, instead of just the saved pages, so Interlisp-D sysout files are very large.

(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.

Interlisp-D 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 Lisp.virtualmem (making it non-continuable). The frequency with which this routine runs is determined (inversely) by:

BACKGROUNDPAGEFREQ [Variable]

This global variable determines how often the routine that writes out dirty pages is run. Initially it is set to 4, so the dirty page routine is run once every 4 times around the idle loop. (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.)

The following function is used to write all of the dirty pages out, to make sure that the current state is not lost if there is a system crash.

(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, dirty pages have been written, and there are few enough dirty pages left to write that a SAVEVM would be quick, SAVEVM will be automatically called. While SAVEVM is being executed, the cursor is changed to a special ''SAV/ING'' cursor. 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 60) 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 rst chance available after the rst dirty page has been written.

18.6 ERROR TYPES

The following additional error types occur in Interlisp- D:

5	FILE SYSTEM ERROR
48	FLOATING UNDERFLOW
49	FLOATING OVERFLOW
50	OVERFLOW
51	ARG NOT HARRAY
52	TOO MANY ARGUMENTS

Interlisp- D allows the user to trap arithmetic exceptions. The action taken when over ow occurs may be set with the function OVERFLOW (page 2.38).

READ-MACRO CONTEXT errors are not generated in Interlisp-D. In the situation where Interlisp-10 would generate the error, the call to READ within the macro will simply return NIL.

18.7 COMPILER

Interlisp-D runs a different instruction set than Interlisp-10, so source files from Interlisp-10 must be recompiled. The default extension (value of COMPILE.EXT) for Interlisp-D compiled files is ''DCOM'' rather than ''COM'' as in Interlisp-10.

The Interlisp-10 compiler translates Lisp source programs into 36-bit PDP-10 instructions. The Interlisp-D compiler compiles Lisp source programs into an 8-bit Lisp instruction set executed by the Xerox 1100 family machines.

In Interlisp-D, block compiling is handled somewhat differently than in Interlisp-10; block compiling provides a mechanism for hiding function names internal to a block, but it does not provide a performance advantage. Block compiling in Interlisp-D 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.

Interlisp-D has an optimizing compiler. Among other optimizations, it performs constant folding. Variables can be declared by the user to be compiler constants using the file package command CONSTANTS (page 11.27), which is syntactically the same as VARS, but additionally informs the compiler that the ''variables'' are constants.

18.8 LINKED FUNCTION CALLS

Linked function calls are not implemented in Interlisp-D. One noticeable result of this is that if you break a function that is used by the system, for example in the READ-EVAL-PRINT loop, you will get unexpected breaks within system code. These extra breaks can be safely exited with OK. To avoid this inconvenience, BREAK the function inside another function, e.g., (BREAK (PRIN1 IN FOO)). (Note: Functions that begin with a backslash (\) are system internal functions and should not be broken or advised.)

18.9 HELPSYS

There is currently no HELPSYS facility in Interlisp-D. There are plans to reimplement a HELPSYS facility eventually.

18.10 OPERATING SYSTEM DEPENDENT FUNCTIONS

Many Interlisp-10 functions are missing from Interlisp-D. An attempt has been made to provide an appropriate implementation for the more useful of these functions, but some simply do not make sense on the Xerox 1100 family machines. For example, there is no such thing as a JSYS. Any function containing a call on JSYS or ASSEMBLE will fail to compile.

The following Interlisp-10 functions are not implemented in Interlisp-D: LISPXSTATS, SUBSYS, GETBLK, RELBLK, ERSTR, GTJFN, OPNJFN, RLJFN, OPENF, JFNS.

The following Interlisp-10 functions are implemented as dummies in Interlisp-D: LISPXWATCH, ADDSTATS, HOSTNAME,, USERNUMBER, HOSTNUMBER, LOADAV. There are communication network analogs of HOSTNAME and HOSTNUMBER called ETHERHOSTNAME and ETHERHOSTNUMBER (page 21.5).

Additional Functions:

(HOSTNAMEP NAME) [Function]
Returns T if NAME is recognized as a valid device or remote file server name at the moment HOSTNAMEP is called.

(DIRECTORYNAMEP DIRNAME HOSTNAME) [Function]
Returns T if DIRNAME is recognized as a valid directory. DIRNAME may include an explicit hostname. If HOSTNAME is supplied, it is used instead. The connected directory and hostname are used as defaults.

(MACHINETYPE) [Function]
Returns the type of machine that Interlisp-D is running on: either DORADO (for the Xerox 1132), DOLPHIN (for the Xerox 1100), or DANDELION (for the Xerox 1108).

(RINGBELLS) [Function]
On the Xerox 1100, this flashes (reverse-videos) the screen several times. On the Xerox 1108, this also beeps through the keyboard speaker.

18.11 IDATE FORMAT

Interlisp-D uses a different time standard than Tenex does. IDATE still has the essential property that (IDATE X) is less than (IDATE Y) if X is before Y, and (IDATE (GDATE N)) equals N. If the particular internal format of the integer date is being used to do arithmetic on dates, the user's programs must be fixed. But in that case the user is already in trouble with Interlisp-10, where the date standard is subtly different between Tenex and Tops20. The most useful property that the three formats have in common is that an internal date can be incremented by an integral number of days by computing as the ''1 day'' constant (which can be evaluated at compile time) the difference between two convenient IDATE's, e.g. (IDIFFERENCE (IDATE " 2-JAN-80 12:00") (IDATE " 1-JAN-80 12:00")).

Currently, the format argument of DATE and GDATE is not supported (an error will occur if the user tries to give one). IDATE now parses most of the date forms allowed in Interlisp-10; e.g., the month can be given numerically, slashes can be used as separators, extra spaces are ignored.

(SETTIME DATE&TIME) [Function]

Sets the internal time-of-day clock. If DATE&TIME = NIL, SETTIME attempts to get the time from the communications net; if it fails, the user is prompted for the time. If DATE&TIME is a string in a form that IDATE recognizes, it is used to set the time.

\TimeZoneComp [Variable]

This variable should be initialized (in {DSK}INIT.LISP) to the time-zone compensation, i.e., the number of hours west of GMT. For the U.S. west coast it is 8. For the east coast it is 5.

18.12 CHARACTER SET

Interlisp-D uses an 8-bit character set whereas Interlisp-10 uses standard 7-bit ASCII. The values returned by CHCON1 range from 0 to 255, and codes in this range are acceptable arguments to CHARACTER and FCHARACTER. Characters 0-127 have their standard ASCII interpretations; characters 128-255 are called ''meta'' characters. Some of the meta characters have printed representations in some fonts (for accents, ligatures, etc.), but most of them will be invisible if printed directly to the screen. Accordingly, the echoing conventions normally defined for control characters have been extended to apply also to meta characters. The echomode of any character may be set by the new function ECHOCHAR (page 6.43). In the original terminal table, the INDICATE character mode is specified for all meta characters, so all meta characters are echoed as a cross-hatch (#) followed by the printed representation corresponding to the 7 rightmost bits of the character. For example, character 129 is echoed as #^A. There is currently no type-in syntax for meta characters.

The CHARCODE function (page 2.12), defined in both Interlisp-D and Interlisp-10, can be useful when dealing with the Interlisp-D character set.

18.13 READ TABLES

In Interlisp-D, all control characters are defined as separator characters in FILERDTBL, so that the font information in files is ignored when files are loaded. Users who run in both Interlisp-10 and Interlisp-D with the same files will want to make the same setting in Interlisp-10's FILERDTBL, in order that files created in one system can be read in the other. The appropriate expression to evaluate, which may be in your Interlisp-10 INIT.LISP file, is:

```
(SETSEPR '(1 2 3 4 5 6 7 9 10 11 12 13 14 15
           16 17 18 19 20 21 22 23 24 25 26)
         1 FILERDTBL)
```

18.14 KEYBOARD INTERPRETATION

In Interlisp-D, keyboard and mouse interpretation is now done entirely by Lisp code, and certain lower level keyboard facilities are therefore available. For each key on the keyboard/mouse there is a corresponding bit in memory that the hardware/microcode turn on and o 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 Ascii codes in the sysbu er, cause interrupts, change the internal shift/control status, or create events to be placed in the mouse bu er.

(KEYDOWNP KEYNAME) [Function]

Used to read the instantaneous state of any key, independent of any bu ering 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 any of the characters on the key-top. The shift keys are named separately as RSHIFT and LSHIFT, space is SPACE, the unmarked keys are BLANK-TOP, BLANK-MIDDLE, and BLANK-BOTTOM, and the mouse buttons are LEFT, MIDDLE, and RIGHT. Paddles on the keyset (not generally available) are named PAD1 through PAD5. Thus (KEYDOWNP 'a) returns T if the ''a'' key is down, (KEYDOWNP 'TAB) returns the state of the TAB key, etc.

(KEYACTION KEYNAME ACTIONS) [Function]

Changes the internal tables that de ne the action to be taken when a key transition is detected by the system keyboard handler. KEYNAME is speci ed as for KEYDOWNP. ACTIONS is a dotted pair of the form (DO WN- ACTION . UP-ACTION), where the acceptable transition actions and their interpretations are:

NIL — Take no action on this transition (the default for up-transitions on all ordinary characters).

a list (CHAR SHIFTEDCHAR LOCKFLA G)

CHAR and SHIFTEDCHAR are either ascii codes or non-digit characters standing for their ascii codes. When the transition occurs, CHAR or SHIFTEDCHAR is transmitted to the system bu er, depending on whether either of the 2 shift keys are down. LOCKFLA G is optional, and may be LOCKSHIFT or NOLOCKSHIFT. If LOCKFLA G 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).

Examples: (a A LOCKSHIFT) and (61Q ! NOLOCKSHIFT) are the initial settings for the down transitions of the ''a'' and ''1'' keys respectively.

1SHIFTUP, 2SHIFTUP, LOCKUP, CTRLUP, METAUP
1SHIFTDOWN, 2SHIFTDOWN, LOCKDOWN, CTRLDOWN, METADOWN

Change the status of the internal ''shift'' ags for the left shift, right shift, shift lock, ctrl, and meta keys, respectively. These shifts a ect the interpretation of ordinary key actions. If either of the shifts is down, then SHIFTEDCHAR s are transmitted. If the lock ag is down, then SHIFTEDCHAR s are transmitted if the key action speci ed LOCKSHIFT. If the control ag is on, then the low-order ve bits are masked out

of the code that would otherwise be transmitted to the system buffer. If the meta flag is down, the high order (8th bit) is turned on as characters are transmitted.

Example: the initial ACTIONS for the left shift key is (1SHIFTUP . 1SHIFTDOWN).

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 KEY ACTIONS SAVECURRENT?) [Function]

KEY ACTIONS 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 KEY ACTIONS.

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 non-NIL, changes the keyboard handler (via KEYACTION) so as to interpret the bottom blank key (''swat'') as a metashift: if a key is struck while 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. Returns previous setting.

18.15 LISPUSERS PACKAGES

Most of the LISPUSERS packages (see page 23.1) are available with the Interlisp-D system as separate loadable packages. The major exception is the HASH package, which is highly machine dependent, and the WHEREIS package which depends on it. EDITA, CJSYS, and many parts of the EXEC package are system-dependent by their very nature, and also are not included. The various network packages are not provided because many of these facilities are integrated into Interlisp-D at a more fundamental level.

Several packages not documented in the Interlisp Reference Manual are available. The list currently includes the following:

GRAPHER — A collection of functions for laying out, displaying, and editing graphs on the Interlisp-D screen.

BROWSER — Modifies the SHOW PATHS command of Masterscope so that the command's output is displayed as an undirected graph. Uses the GRAPHER package.

EVALSERVER — Provides a set of routines to facilitate communication, over an Ethernet, between two or more Xerox 1100s running Interlisp-D.

HISTMENU	Provides a simple way to access the Interlisp history list using a menu.
SAMEDIR	This package advises MAKEFILE to notify the user if it appears that a le is being written onto a directory other than the one it came from, allowing the user to halt the process.

18.16 FILE SYSTEM

Typically, the most machine- dependent part of any computer language implementation is the I/O system. Regardless of e orts to create consistant interfaces, the fact remains that di erent physical machines o er di erent disks, printers, etc., and languages have to be extended to take advantage of these. In the case of implementing Interlisp on the Xerox 1100 family machines, the biggest change was the addition of facilities for using the high- resolution display, described elsewhere. Other changes have had to be made to accomodate using les on a local disk or on a le server, and sending les to remote printers. Every e ort has been made to keep these interfaces compatible with Interlisp- 10 conventions, to reduce the amount of work necessary when transferring programs. However, in some situations the user may wish to take advantage of the special extensions o ered by Interlisp- D.

This section contains information about a variety of extensions to Interlisp- D that accomodate the di erent I/O environment.

18.16.1 File Names

‘‘Full’’ le names inside of Interlisp- D look just like Tenex le names, except that all full le names begin with a device/host name (in braces) to identify the machine (or pseudo- machine) on which the le resides. Files on the local disk belong to device/host DSK, e.g. {DSK}FOO.BAR;3. PACKFILENAME and UNPACKFILENAME are still the appropriate way for programs to manipulate lenames. The device/host of a le may be accessed using the new eld name HOST.

On Xerox 1100s and Xerox 1132s, Interlisp- D can access partitions other than the one which was booted. If the other partition is password- protected, Interlisp insists on the correct password before accessing any les. Partitions are denoted by {DSK1} for Partition 1, {DSK2} for Partition 2, etc. DIR, DIRECTORY, etc. all work for other partitions. Currently, SYSOUT does not work for partitions other than the default.

18.16.2 Renaming Files

Interlisp- D implements (RENAMEFILE OLD NEW) merely by copying OLD to NEW and then deleting OLD . While this is quite general (and even allows one to rename les from the local disk or one le server to another), it is slower than the Interlisp- 10 RENAMEFILE operation. It also, in the case of renaming a local disk le, requires that the local disk have enough room to hold the copy of the le.

18.16.3 End Of Line Convention

Interlisp- D uses a di erent representation for *end of line* both internally and on les. Internally, *end of line*

is represented by the carriage return character (15Q), whereas the internal representation in Interlisp-10 is the EOL character (37Q). The CHARCODE macro (page 2.12) is the appropriate way to code programs to be independent of the EOL convention: in all systems (CHARCODE EOL) is always the appropriate end-of-line character. (CHARCODE CR) and (CHARCODE TENEXEOL) provide the system-dependent character codes. Interlisp-D also interprets a carriage return/line feed sequence in a file as an end-of-line and reads it as a carriage return. TERPRI generates two characters in Interlisp-10, but only one in Interlisp-D.

18.16.4 Using Files with Processes

Currently, Interlisp-D does not provide interlocks to keep multiple processes from trying to access the same file. Therefore, the user has to be careful not to have two processes manipulating the same file at the same time. For example, it will not work to have one process TCOMPL a file while another process is running LISTFILES on it.

18.16.5 Miscellaneous File Manipulation

(COPYFILE FROMFILE TOFILE) [Function]
Copies a file to a new file. The source and destination may be any servers/devices. COPYFILE attempts to preserve the TYPE and CREATIONDATE where possible.

(DISKFREEPAGES _ _) [Function]
Returns an estimate of the number of pages free on the local disk (current partition). This number is only a ''hint'', but is usually quite accurate.

(DISKPARTITION) [Function]
Returns the number of the current partition (1 or 2 on Xerox 1100, 1-5 on Xerox 1132).

18.16.6 Connecting to Directories

As in Interlisp-10, Interlisp-D has a notion of a ''connected'' directory, which is used as the default when you give a filename lacking an explicit device/host (and directory). The default is changed by using the programmer's assistant command CONN.

CONN {DEVICE/HOST}<DIRECTORY> [Prog. Asst. Command]
Either part of the argument is optional; if the directory is omitted, the default for devices that have directories is the value of (USERNAME); if the host is omitted, connection will be made to another directory on the same host as before. If CONN is given with no arguments, connects to the value of LOGINHOST/DIR.

Note that CONN does not require or provide any directory access privileges, as does the command of the same name in Interlisp-10. Access privileges are checked when a file is opened.

(CNDIR HOST/DIR) [Function]
Programmatic form of CONN. Connects to the directory HOST/DIR. Returns the

fullname of the now-connected directory.

(/CNDIR HOST/DIR) [Function]
Undoable form of CNDIR. CONN is implemented via /CNDIR.

LOGINHOST/DIR [Variable]
CONN with no argument connects to the value of the variable LOGINHOST/DIR, initially {DSK}, but usually reset in the user's greeting le.

(DIRECTORYNAME FLG STRPTR) [Function]
Similar to Interlisp-10 USERNAME. If FLG is T, returns the currently connected host and directory name. If FLG is NIL, returns the value of LOGINHOST/DIR. If STRPTR is T, the value is returned as an atom, otherwise it is returned as a string.

DIRECTORIES [Variable]
Global variable containing the list of directories searched (in order) by SPELLFILE and FINDFILE (page 15.20) when not given an explicit DIRLST argument. In this list, the atom NIL stands for the login directory (LOGINHOST/DIR), and the atom T stands for the currently connected directory.

18.16.7 Binary I/O

Interlisp-D supports a datatype called a STREAM, whose basic operations are ''input'' and ''output''. They provide an e cient handle to an open le. All I/O functions that currently refer to les (e.g., PRINT, PRIN1, COPYBYTES, FULLNAME) will also accept streams, and will operate slightly more e ciently on them. In addition, the following two functions provide binary input and output on streams:

(BIN STREAM) [Function]
Returns the next byte from STREAM ; thus, this operation is similar to (CHCON1 (READC STREAM)). BIN is a very e cient (microcoded) operation.

(BOUT STREAM BYTE) [Function]
Outputs a single 8-bit byte to STREAM , i.e., similar to (PRIN3 (CHARACTER BYTE)).

In addition, the following function coerces les to streams:

(GETSTREAM FILE ACCESS) [Function]
Takes a designator which can be used as a '' le'' argument (e.g., a full/partial le name, a display stream, window, etc.) and returns the corresponding stream. If given a stream will merely return it. ACCESS is interpreted the same as in OPENP (page 6.2).

BIN and BOUT will also accept a le designator, in which case they coerce it to a stream via GETSTREAM. However, BIN executes in microcode only when given a stream directly.

18.16.8 Temporary Files and the CORE Device

The local DSK device and most le servers do not support the temporary or scratch les that are available

in Interlisp-10. Files that are created do not disappear when some later event such as logout occurs and instead must be deleted by specific action on the part of the user. For this reason, the ;S and ;T suffixes in file names are simply ignored when output is directed to a particular host or device.

However, Interlisp-D does support a notion of core-resident files, and in many cases these provide a reasonable substitute for Interlisp-10 scratch files. Core-resident files are on the device CORE (e.g. {CORE}<FOO>FIE.DCOM;5). The directory for this device and all files on it are represented completely within the user's 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.

In Interlisp-D, the function PACKFILENAME is defined to default the device name to CORE if the file has the TEMPORARY attribute and no explicit host is provided.

Interlisp-D is initialized with the single core-resident device CORE, but the function COREDEVICE may be used to create any number of logically distinct core devices.

(COREDEVICE NAME) [Function]
Creates a new device for core-resident files and assigns NAME as its device name. Thus, after performing (COREDEVICE 'FOO), one can execute (OUTFILE '{FOO}BAR) to open a file on that device.

If the directory information associated with CORE devices is not needed, the device NODIRCORE can be used to open core-resident files which ''disappear'' when they are closed. Note that {NODIRCORE} files do not have names, so the only way to manipulate them is to pass around the value that OPENFILE returned when the file was opened.

18.16.9 Floppy Disks on the Xerox 1108

Interlisp-D on the Xerox 1108 can access the built-in floppy disk drive as device {FLOPPY}. The floppy format is compatible with the Pilot floppy disk format.

18.16.10 Page Mapping

Interlisp-D implements the page-mapping primitives of Interlisp-10 with some notable differences that might require major reworking of programs that rely on these facilities (see page 14.17). The major difference is that an Interlisp-D page contains 256 16-bit words, rather than the 512 36-bit words of Interlisp-10. A given page number or file address for MAPPAGE or MAPWORD will correspond to a very different number of bits from the beginning of the file, and WORDCONTENTS and SETWORDCONTENTS move smaller amounts of information. A second difference is that buffers are completely integrated into the Interlisp-D storage management system so that a page is guaranteed to be locked down as long as the user holds a pointer to it. The functions LOCKMAP and UNLOCKMAP are therefore unnecessary, but for compatibility are defined with dummy definitions.

18.17 FILE SERVERS

A file server is a shared resource on a local communications network which provides large amounts of

le storage. Di erent le servers honor a variety of access protocols. In order to support full Lisp I/O, a le server must provide a random access protocol. One such protocol is Leaf. It has been integrated into the Interlisp- D le system to allow les on a le server to be treated in much the same way les are accessed on the local disk. Except where noted in this section, the standard le operations (OPENFILE, INFILEP, CLOSEF, etc.) all work for remote les. This section explains how to make use of remote les and what di erences exist between them and other les.

18.17.1 File Server File Names

The full name of a le on a le server host includes the name of the host in braces, and a directory speci cation in angle brackets, e.g., {PHYLUM}<LISP>FOO.DCOM;3 . These names are not necessarily the syntax by which the actual device/server knows the les (e.g. some le servers use ‘‘!’’ instead of ‘‘;’’), but Lisp presents a uniform set of naming conventions.

The user can ‘‘connect’’ to a directory on a le server using the CONN command (page 18.11), after which any lename supplied that does not include the host name and/or directory will use the ‘‘connected’’ host and/or directory. Speci cally, if the host is omitted, then the connected host is used, and if the directory is also omitted, the connected directory is used as well. If an explicit host is supplied, no defaulting of the directory occurs.

Interlisp supports a prelimi nary version of NS ling to Xerox 8030 le servers (see page 21.13). Any device with a colon in its name is presumed to be accessible with NS protocols rather than PUP, e.g., {STARFILE:}. The general format of NS leserver device names is {SERVER :DOMAIN :ORGANIZATION }; the device speci cation for an 8000-series product must contain the ClearingHouse domain and organization, but if not supplied directly, then they are obtained from the defaults, which themselves are found by a search for the nearest ClearingHouse. NS le servers are modeled after the Star world, and have ‘‘File Drawers’’ rather than directories; ‘‘File Folders’’ are like sub-directories. The functions DIRECTORY, FILEBROWSER, INFILE, COPYFILE, LOAD, and MAKEFILE are working now with NS le servers.

NETWORKOSTYPES [Variable]

Files servers on di erent machines have di erent login protocols, le name formats, etc. For proper service from le servers other than Xerox le servers, the user should add entries to the association-list NETWORKOSTYPES associating the host name (all uppercase) with its operating system type, currently one of TENEX, TOPS20, UNIX, or VMS. For example (ADDTOVAR NETWORKOSTYPES (MAXC2 . TENEX)) will inform Interlisp that the le server MAXC2 is a TENEX le server.

18.17.2 Logging In

Most le servers require a user name and password for access. When a le server requests this information, Interlisp- D rst gives the name and password from the Alto Executive. If the le server doesn’t recognize that name/password, Interlisp- D prompts the user for a name and password to use. It suggests a default name (the one on the disk), which the user can accept by typing a space, or replace by typing a new name or backspacing over it. Interlisp- D saves names and passwords for each host, so the user can login to di erent le servers using di erent names.

(LOGIN HOSTNAME _ _ _) [Function]

Forces Interlisp- D to ask for the login name and password to be used when accessing host HOSTNAME . Any previous login information for HOSTNAME is

overriden. If HOSTNAME is NIL, it overrides login information for all hosts. Password information vanishes when LOGOUT, SYSOUT, or MAKESYS is executed. Returns the login user name.

18.17.3 Abnormal Conditions

If Interlisp-D tries to access a file and does not get a response from the file server in a reasonable 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 CTRL-E or similar interrupt aborts out of this state.

If the file server crashes but is restarted before the user attempts to do anything, file operations will usually proceed normally, except for a brief pause while Interlisp-D tries to reestablish any connections it had open before the crash. It will inform the user of any problems that arise in so doing. The most likely problem occurs when a file has been opened for output but has not yet been written to (or not enough has been written so that Interlisp-D has written to the file server). In this case the file server will think the file is not there when Interlisp-D tries to reestablish the connection. A similar situation arises if the system has been idle (or at least has not accessed the file server) for a sufficiently long period. In this case, the file server will time out the connection. Normally, Interlisp-D will attempt to recover gracefully as described above.

LOGOUT closes any Leaf connections that are currently open. On return, it attempts to reestablish connections for any files that were open before logging out. If a file has disappeared or been modified, Interlisp-D reports this fact.

If it is desired to break the Leaf connection without logging out, call (BREAKCONNECTION HOST). Any subsequent reference to files on that host will reestablish the connection. The main reason for doing this occurs if Interlisp-D is interrupted while a file is being opened, leaving the file server thinking the file is open and Lisp thinking it is closed, and then getting a file busy when Interlisp-D next tries to open it.

On rare occasions, the Ethernet may appear completely unresponsive, due to Interlisp having gotten into a bad state. Typing (RESTART.ETHER) will reinitialize Lisp's Ethernet driver(s), just as when the Lisp system is started up following a LOGOUT, SYSOUT, etc (see page 21.15)

18.17.4 Caveats

Leaf does not currently support directory enumeration except for one minor case (in the version field). Hence, DIRECTORY or FILDIR cannot be used on a Leaf file server to get a list of files.

INFILEP and GETFILEINFO currently 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. This differs from the operation of DSK, and from Interlisp-10 file operations.

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 background''. It is therefore very important that the user exits Interlisp via (LOGOUT), or (LOGOUT T), rather than boot the machine or exit via Raid.

18.17.5 New Functionality

Certain le servers treat text and binary les di erently. Files on le servers can have the attribute TYPE, with value TEXT or BINARY, for use with GETFILEINFO and SETFILEINFO. The le type defaults to the value of DEFAULTFILETYPE, initially TEXT. OPENFILE accepts (TYPE TEXT) or (TYPE BINARY) as an element of its argument MA CHINE.DEPENDENT.P ARAMETERS .

Another allowed element of MA CHINE.DEPENDENT.P ARAMETERS is DON'T.CHANGE.DATE, which means not to change the le's creation date when a le is opened (meaningful only for les being opened for output).

Interlisp- D includes an implementation of the PupFtp protocol, which supports transferring les sequentially only. In those cases where sequential access (as opposed to random access) to a le is appropriate, the use of PupFtp generally results in considerable speed improvement over Leaf, particularly for writing les on a Xerox IFS. The system tries to use PupFtp where possible for SYSOUT and for the destination le of a COPYFILE. One can indicate that a le is going to be accessed only sequentially by including the keyword SEQUENTIAL in the list of MA CHINE.DEPENDENT.P ARAMETERS passed to OPENFILE; the PupFtp will be used, if possible. If for some reason your le server supports PupFtp but you do not wish COPYFILE or SYSOUT to use it, you can set the internal variable \FTPAVAILABLE to NIL.

18.18 HARDCOPY FACILITIES

Note: The following implementation of hardcopy facilities is subject to change.

Interlisp- D includes facilities for generating hardcopy in both ''Press'' and ''Interpress'' formats. ''Press'' is a le format used for communicating documents to laser Xerographic printers called ''Dover'' (at MIT, Stanford, and CMU) or ''Penguin'' (everywhere else). ''Interpress'' is a Xerox standard format used by the 8044 printer and other Network System printers. The hardcopy functions below will generate Press or Interpress output depending on the setting of the function PRINTERMODE:

(PRINTERMODE x) [Function]

Sets the type of printing le format generated by LISTFILES, HARDCOPYW, and printer devices (see PRINTERDEVICE, below). If x is PRESS, the Press le format is used. If x is INTERPRESS, the Interpress le format is used.

Currently, the hardcopy interface is not smart enough to infer the printer mode from a previously formatted le or the name of a printing host. If the user wants to print a previously formatted Press or Interpress le, the printing mode must be set correctly.

(PRINTINGHOST _) [Function]

The function PRINTINGHOST is used to nd the name of the local printer.

For (PRINTERMODE 'PRESS), this merely returns the value of the variable DEFAULTPRINTINGHOST, which is usually set by an entry in the site greeting le (see page 14.5).

For (PRINTERMODE 'INTERPRESS), this returns the value of the variable NS.DEFAULT.PRINTER if non-NIL, otherwise it returns the first local printer found in the closest clearinghouse (see page 21.11).

The function LISTFILES1 is used by LISTFILES to send a single file to a hardcopy printing device. Interlisp-D is initialized with LISTFILES1 defined to call EMPRESS in Press mode or NSPRINT (page 21.11) in Interpress mode. These functions convert a file to Press or Interpress format and send it to a printing server. The ''default'' site greeting file delivered with the Xerox 1100 redefines LISTFILES1 as a no-op.

(EMPRESS FILE qCOPIES HOST HEADING qSIDES) [Function]

The function EMPRESS causes qCOPIES copies of the file FILE to be sent to the printer HOST. If HOST is NIL, the value of (PRINTINGHOST) is used. qSIDES specifies one- or two-sided printing; may be 1 or 2 (if HOST is capable of duplex printing) or T (meaning to use the printer's default); defaults to the value of EMPRESS#SIDES, initially T.

If FILE is a Press or Interpress format file, it is transmitted directly. Otherwise, it is converted by calling the function MAKEPRESS (called with FONTS = NIL and the same HEADING).

EMPRESS.SCRATCH [Variable]

EMPRESS constructs scratch press files on the {CORE} device for small files. If the number of disk pages of the source file is larger than the limit set by the first element of the list EMPRESS.SCRATCH, an alternate scratch file, specified by the second element of EMPRESS.SCRATCH, is used. EMPRESS.SCRATCH is initialized to (30 {DSK}EMPRESS.SCRATCH).

(MAKEPRESS FILE OUTFILE FONTS HEADING TABS) [Function]
(MAKEINTERPRESS FILE OUTFILE FONTS HEADING TABS) [Function]

These functions produce a Press or Interpress file named OUTFILE from the ASCII file FILE. If OUTFILE is NIL, it defaults to the same file name as FILE, with extension Press or Interpress.

These functions interpret character sequences beginning with control-F (character code 6) as special formatting instructions. If the code of the next character is a valid font number, then the formatting sequence indicates a change to that font. The correspondence between font numbers and fonts is specified by entries on the list FONTS or, if FONTS is NIL, the current font profile list (see page 6.55). Each entry is of the form (FONT/CLASS FONT-NUMBER DISPLAY-FONT PRESS-FONT). For example, the entry (DEFAULTFONT 1 (GACHA 10) (GACHA 8)) indicates that GACHA 8 will be used in press files for font 1 which will be represented on the display as GACHA 10. HEADING is a string that is printed as a heading on each page. If HEADING is NIL, the file's name and creation date will be used.

These functions also allow absolute tab stops to be specified. If the control-F is followed by a control-T, the code of the character after that is interpreted as an absolute tab stop number. The corresponding entry on the list TABS, or PRESSTABSTOPS if TABS is NIL, is taken as the number of mills from the left margin at which printing on the current line will continue. PRESSTABSTOPS is initially (8000).

FONTWIDTHSFILES [Variable]

Value is a le name or a list of le names to be searched for information about the widths of characters in particular fonts. This variable should be initialized in the site greeting le.

(HARDCOPYW WINDOW/BITMAP/REGION FILE HOST SCALEFACTOR ROTATION) [Function]

Creates bitmap hardcopy and optionally sends it to a printer. WINDOW/BITMAP/REGION can either be a WINDOW (open or closed), a BITMAP, or a REGION (interpreted as a region of the screen). If NIL, the user is prompted for a region using GETREGION (page 19.37) in a manner which ''defaults'' to the whole screen.

The logic of defaulting is complex and follows:

FILE, if supplied, will be used as the name of the le for output. If HOST is NIL, then if FILE was given, no printing is performed, else if FULLPRESSPRINTER is non-NIL, then output is sent to that printer, else output is sent to the value of (PRINTINGHOST). To save an image on a le without printing it, perform (HARDCOPYW IMAGE FILE).

SCALEFACTOR is a reduction factor. Only SCALEFACTOR =1 can be printed on Dover and Penquin printers. SCALEFACTOR defaults according to the size of the image, the size of a page, and the parameters HOST, FILE, and FULLPRESSPRINTER in a complex but appropriate manner.

ROTATION, which can be one of 0, 90, 180, 270 (default 0) speci es how the bitmap image should be rotated on the printed page. This may not be supported by some printers.

Note that ''Hardcopy'' in the background menu merely performs (HARDCOPYW), which sends an image of region user selects to the default printer. Hardcopy in the paint menu performs (HARDCOPYW WINDOW), which sends an image of window to the default printer.

(PRESSFILEP FILE) [Function]

Returns (FULLNAME FILE) if FILE is a Press le, NIL otherwise.

Hardcopy output may also be obtained by writing a le on the printer device LPT, e.g. (COPYFILE 'FOO '{LPT}). When a le on this device is closed, it is converted to Press or Interpress format (if necessary) and sent to the default printer. Thus, {LPT} acts like the device LPT: in Interlisp-10. Printer devices can be de ned for other network printer hosts with the following function:

(PRINTERDEVICE NAME) [Function]

De nes the network printer host NAME to be a printer device treated like LPT. For example, if (PRINTERDEVICE 'YODA) is executed, then (COPYFILE 'FOO '{YODA}) will transmit FOO to the printer named YODA.

18.19 PERFORMANCE CONSIDERATIONS

Most Interlisp-D users will have experience using Interlisp-10. Although Interlisp-D is completely upward

compatible with Interlisp-10, there are differences in the exact implementation which may influence the performance of applications programs. This chapter contains a collection of notes which may help the user improve the performance of Interlisp-D programs.

18.19.1 Variable Bindings

A major difference between Interlisp-10 and Interlisp-D is the method of accessing free variables. Interlisp-10 uses what is called ''shallow'' binding. Interlisp-D uses what is called ''deep'' binding.

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 (atoms) 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 atom 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).

A deep binding scheme was choosen for Interlisp-D because of the working set considerations and the speed of context switching, which we expected to use heavily when processes were added. The free variable lookup routine was microcoded, thus greatly reducing the search time. In the benchmarks we performed, 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 (page 11.25). Its form is (GLOBALVARS VAR_1 ... VAR_N).

Another way of improving performance is to declare variables as local within a function. Normally, all variables bound within a function have their names put on the stack, and these names are scanned during free variable lookup. If a variable is declared to be local within a function, its name is not put on the stack, so it is not scanned during free variable lookup, which may increase the speed of lookups. The compiler can also make some other optimizations if a variable is known to be local to a function.

A variable may be declared as local within a function by including the form (DECLARE (LOCALVARS VAR_1 ... VAR_N)) following the argument list in the definition of the function. Note: 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.

18.19.2 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 get filled up, and the computation would come to a halt. Actually, long before this would 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. This problem (''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 frought with pitfalls. 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. Interlisp-10 uses the traditional mark-and-sweep garbage collection algorithm, which identifies ''garbage'' data by ''walking'' through and ''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 simply 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 time for scanning the reference count tables is very nearly constant (about 0.2 seconds on the Xerox 1100); the sweep time then is this small value, 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 on page 18.2. 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.

18.19.3 Datatypes

If an applications program uses data structures that are large (more than 8 fields) and that are used a lot, there are several advantages to representing them as user `DATATYPE`s rather than as `RECORD`s. The primary advantage is increased speed: accessing and setting the fields of a `DATATYPE` can be significantly faster than walking through a `RECORD` list with repeated `CAR`s and `CDR`s. Also, compiled code for referencing user `DATATYPE`s is usually smaller. Finally, by reducing the number of objects created (one `DATATYPE` object against many `RECORD` list cells), this can reduce the expense of garbage collection.

For code that has been written using the record package's `fetch`, `replace`, and `create` operations, changing from `RECORD`s to `DATATYPE`s only requires editing the record declaration (using `EDITREC`) to replace declaration type `RECORD` by `DATATYPE`, and recompiling.

18.19.4 Incomplete Filenames

There is a significant problem in Interlisp-D (and in Interlisp-10) with respect to using incomplete filenames. Whenever an I/O function is given an incomplete filename (one which doesn't have the device/host, directory, name, extension, and version number all supplied), the system has to convert it to a complete filename, by supplying defaults and searching through directories (which may be on remote file servers). Currently, work is being done on speeding up the filename-completion process, but in any case it is much faster to convert an incomplete filename once, and use the complete filename from then on. For example, suppose a file is opened with `(SETQ FULLNAME (OPENFILE 'MYNAME 'INPUT))`. After doing this, `(READC 'MYNAME)` and `(READC FULLNAME)` 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.

18.19.5 Turning O the Display

Maintaining the video image on the screen uses about 30% of the cpu cycles (on the Xerox 1100), so turning o the display will improve the speed of compute-bound tasks. When the display is o, 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 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 following functions are provided to turn the display o:

(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.

18.19.6 Gathering Statistics

Interlisp-D has an extended set of statistics-gathering tools. An extended version of the TIME function is provided:

(TIMEALL TIMEF ORM q TIMES TIMEWHA T INTERPFL G _) [NLambda Function]
Largely subsumes the function TIME. Evaluates the form TIMEF ORM and prints statistics on time spent in various categories (elapsed, keyboard wait, swapping time, gc) and datatype allocation.

For more accurate measurement on small computations, q TIMES may be speci ed (its default is 1) to cause TIMEF ORM to be executed q TIMES number of times. To improve the accuracy of timing open-coded operations in this case, TIMEALL compiles a form to execute TIMEF ORM q TIMES number of times (unless INTERPFL G is non-NIL), and then times the execution of the compiled form. The compilation is with optimizations o to avoid constant folding.

TIMEWHA T exists largely for compatibility with TIME; it restricts the statistics to speci c 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.

The value of TIMEALL is the value of the last evaluation of TIMEF ORM .

The Interlisp-D system has a facility for gathering very low-level statistics on function call and return. It is conceptually like performing a BREAKDOWN on every function in the world. The system designers regularly use this facility to determine where time is being spent in suspect computations, suggesting which parts of the system code deserve optimizing.

(DOSTATS FORM TITLE _ _ _) [Function]
Collects statistics of the evaluation of FORM and produces a listing of the results. TITLE , if supplied, will appear in the heading of the listing.

INTERLISP-D SPECIFICS

Performing a statistics run consists of three phases:

Gathering

The microcode is instructed to emit a statistics event for every function call and return that is executed, and FORM is evaluated. These events are collected on a file for the next phase (the name of the file is {DSK}XXX.STATS, where XXX = (CAR FORM)). Currently the file must reside on {DSK}, so be sure you have a lot of space. Even seemingly short computations can generate large numbers of function call/return events. If your disk fills up, Lisp may not recover gracefully (it usually falls into SWAT).

Analysis

The statistics file is read in. For each event, a counter associated with the indicated function is incremented by the amount of time spent in the function. The analysis also records who called which functions, how often, and with how many arguments. This is by far the longest phase.

Summarizing

The results of the analysis are used to produce a listing that shows each of the functions called, sorted by their contribution to the total time, and a cross-reference of who called whom. The listing is put on a file XXX.PRINTOUT on the connected directory and also shipped to your local printer.

Excerpts from a sample statistics printout are shown below, with commentary. The form is (RECLAIM), which was fairly brief in this case.

Notes

The times shown in the printout are for time spent in a single function; there is no cumulative time measurement. The percentages should thus add up to 100%. If FOO calls FIE, the time spent inside FIE is charged to FIE only, not to FOO as well.

The times recorded are of the right order of magnitude, and can be compared to each other, but should not be taken literally, as they are inflated by the overhead of recording each call and return event. The total elapsed time for the evaluation phase is much larger still, being dominated by the time to dump the statistics to disk, but this part of the time is filtered out in the analysis.

```
Statistics from file: {DSK}RECLAIM.STATS;1

measuring: evaluation of
FORM = (RECLAIM)

Computation run on Dolphin serial #237 with 2304 pages of memory.
Versions:  Ram=7401(17,1)  Bcpl=17400(37,0)  Lisp=106000(214,0)
           (Internal version numbers of microcode, Lisp.run, Lisp.sysout)

Unrecognized events: NIL (everything was okay)

Values from MiscStats (times in msecs):
      SWAPWAITTIME             6137
      PAGEFAULTS                 58
      GCTIME                  27392

Not Windowing
```

```
Filtering out \StackOverflow, \NWWInterrupt, \PageFault, \StatsOverflow
```
(time for these functions measured separately)

```
Ignoring time for GETKEYS, \GETKEY, WAITFORINPUT, DISMISS, GATHERSTATS,
                  \GATHERSTATS, RAID
```
(time for these functions ignored completely)

```
Function timings:                       #ofCalls    PerCall
```
total time spent in function (microseconds)
| percentage of total analyzed time spent in function
| | function name. Number of arguments in brackets
| | | number of calls recorded to this fn
| | | | avg time per call (microseconds)

```
1746426  36.08%  \GCMAPTABLE [1]          524       3332
1104420  22.81%  \GCMAPSCAN [0]            1    1104420
 794862  16.42%  \HTFIND [2]             1236        643
 461194   9.52%  \FREELISTCELL [1]       2044        225
 457537   9.45%  \GCRECLAIMCELL [1]      1533        298
  77437   1.59%  \GCMAPUNSCAN [0]           1      77437
  52907   1.09%  RELEASINGVMEMPAGE [1]     30       1763
  47308   0.97%  \GCSCANSTACK [0]           1      47308
  45365   0.93%  FINDPTRSBUFFER [2]        30       1512
   9218   0.19%  \ADDBASE [2]              31        297
   7618   0.15%  CREATECELL [1]            18        423
   7428   0.15%  \INSERTBLOCK [1]          31        239
   6856   0.14%  \RECLAIMARRAYBLOCK [1]    31        221
  21597   0.44%  for 18 entries not shown
                   (functions contributing less than .1% are omitted)
4840173          Total for 31 entries    5511


Function timings: Filtered out fns   #ofCalls   PerCall
```
(times for functions whose contribution was omitted from the analysis above)

```
20225828  70.32%  Subr.\StatsOverflow [0] 413      48972   (stats overhead)
 6900042  23.99%  Subr.\PageFault [1]      58     118966   (pagefault activity)
 1635737   5.68%  Subr.\NWWInterrupt [0]  762       2146   (periodic service)
28761607          Total for 3 entries    1291


Function timings: Alphabetic         #ofCalls   PerCall
```
(listing as above, but including all functions, and sorted alphabetically)

. . .

```
Call Information:
```
(Alphabetic listing of functions, with calls and callers information)

(number of calls in parentheses)

```
CLOCK
          Calls:   MAKENUMBER (8), \SLOWIPLUS2 (6), CLOCK (2),
                   CREATECELL (2), CLOCK0 (2), \SLOWIDIFFERENCE (2)
          Callers: \DORECLAIM (2), CLOCK (2)
CLOCK0
          Callers: CLOCK (2)
CREATECELL
          Calls:   \HTFIND (1)
          Callers: MAKENUMBER (16), CLOCK (2)

      . . .
```

18.20 THE INTERLISP-D PROCESS MECHANISM

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 adress 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.

Note: `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`.

18.20.1 Creating and Destroying Processes

(ADD.PROCESS FORM PROP$_1$ VALUE$_1$ PROP$_N$ VALUE$_N$) [NoSpread Function]

Creates a new process evaluating FORM, and returns its process handle. The process's stack environment is the top level, i.e., the new process does not have access to the environment in which ADD.PROCESS was called; all such information must be passed as arguments in FORM. The process runs until FORM returns or the process is explicitly deleted. An untrapped error within the process also deletes the process (unless its RESTARTABLE property is T), in which case a message is printed to that effect.

The remaining arguments are alternately property names and values. Any property/value pairs acceptable to PROCESSPROP may be given, but the following two are directly relevant to ADD.PROCESS:

NAME Value should be a litatom; if not given, the process name is taken from (CAR FORM). ADD.PROCESS may pack the name with a number to make it unique. This name is solely for the convenience of manipulating processes at Lisp typein; e.g., the name can be given as the PROC argument to most process functions, and the name appears in menus of processes. However, programs should normally only deal in process handles, both for efficiency and to avoid the confusion that can result if two processes have the same defining form.

SUSPEND
If the value is non-NIL, the new process is created but then immediately suspended; i.e., the process does not actually run until woken by a WAKE.PROCESS (below).

(PROCESSPROP PROC PROP NEWVALUE) [NoSpread Function]

Used to get or set the values of certain properties of process PROC, in a manner analogous to WINDOWPROP. If NEWVALUE is supplied (including if it is NIL), property PROP is given that value. In all cases, returns the old value of the property. The following properties have special meaning for processes; all others are uninterpreted:

NAME Value is a litatom used for identifying the process to the user.

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.

FORM Value is the Lisp form used to start the process (readonly).

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 (page 18.30).

INFOHOOK
Value is a function or form used to provide information about the process, in conjunction with the process status window (page 18.36).

WINDOW
Value is a window associated with the process, the process's ''main'' window. Used in conjunction with switching the tty process (page 18.33).

TTYENTRYFN
Value is a function that is applied to the process when the process is made the tty process (page 18.33).

TTYEXITFN
Value is a function that is applied to the process when the process ceases to be the tty process (page 18.33).

(THIS.PROCESS) [Function]
Returns the handle of the currently running process, or NIL if the Process world is turned o .

(DEL.PROCESS PR OC _) [Function]
Deletes process PR OC . PR OC may be a process handle (returned by ADD.PROCESS), or its name. Note that if PR OC 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. Note that 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.

18.20.2 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 (in which case BLOCK is very much like DISMISS), or T, meaning wait forever (until explicitly woken). Alternatively, TIMER can be given as a millisecond timer (as

returned by SETUPTIMER) 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.

(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.

(PROCESS.APPLY PROC FN ARGS WAITFORRESULT) [Function]

Performs (APPLY FN ARGS) in the stack context of PROC. Note same warnings as with PROCESS.EVAL.

18.20.3 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 noti ed, 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 page 14.11).

(NOTIFY.EVENT EVENT ONCEONL Y) [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 ONCEONL Y is true, only runs the rst 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 noti cation 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 noti ed event.* In particular, the completion of PROCESS.EVAL and related operations in e ect 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-de ned events, used with the network code. Each Pup and NS socket has associated with it an event that is noti ed when a packet arrives on the socket; the event can be obtained by calling (PUPSOCKETEVENT PUPSOCKET) or (NSOCKETEVENT NSOCKET), respectively.

18.20.4 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 the user and associated with (e.g., stored in) some shared structure that is to be protected from simultaneous access. To access the structure, a program waits for the lock to be free, then takes ownership of the lock, accesses the structure, then releases the lock. The functions and macros below are used:

(CREATE.MONITORLOCK NAME _) [Function]
Returns an instance of the MONITORLOCK datatype, to be used as the lock argument to functions listed below. NAME is arbitrary, and is used for debugging or status information.

(WITH.MONITOR LOCK . FORMS) [Macro]

Evaluates (PROGN . FORMS) while owning LOCK. Value is the last of FORMS. 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 is a noop.

(WITH.FAST.MONITOR LOCK . FORMS) [Macro]

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

Programming restriction: the evaluation of FORMS must not error (the lock would not be released). This construct is mainly useful when FORMS is 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 FooLock
   (until conditionOfFoo
    do (MONITOR.AWAIT.EVENT FooLock EventFooChanged timeout))
       operateOnFoo)
```

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, the user 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) [Function]
Releases LOCK if it is owned by the current process, and wakes up the next process, if any, waiting to obtain the lock.

When a process is deleted, any locks it owns are released.

18.20.5 Global Resources

The biggest source of problems in the multi-processing environment is the matter of global resources. Two processes cannot both use the same global resource if there can be a process switch in the middle of their use (currently this means calls to BLOCK, but ultimately with a preemptive scheduler means anytime). Thus, user code should be wary of its own use of global variables, if it ever makes sense for the code to be run in more than one process at a time. ''State'' variables private to a process should generally be bound in that process; structures that are shared among processes (or resources used privately but expensive to duplicate per process) should be protected with monitor locks or some other form of synchronization.

Aside from user code, however, there are many *system* global variables and resources. Most of these arise historically from the single-process Interlisp-10 environment, and will eventually be changed in Interlisp-D to behave appropriately in a multi-processing environment. Some have already been changed, and are described below. Two other resources not generally thought of as global variables the keyboard and the mouse are particularly idosyncratic, and are discussed in the next section.

The following resources, which are global in Interlisp-10, are allocated per process in Interlisp-D: primary input and output (the streams a ected 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 les. Thus, each process can print to its own primary output, print to the terminal, read from a di erent 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 speci ed 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 PROC) [Function]
Returns T if the process PROC has a tty window; NIL otherwise. If PROC is NIL, it defaults to the current process.

Other system resources that are typically changed by RESETFORM, RESETLST, 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 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).

18.20.6 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 the 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 on page 18.35.

18.20.6.1 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) [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.

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.

Switching the TTY Process

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 su ces 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, speci cally 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 (WINDO W). WINDOWENTRYFN defaults to GIVE.TTY.PROCESS:

(GIVE.TTY.PROCESS WINDO W) [Function]
If WINDO W has a PROCESS property, performs (TTY.PROCESS (WINDOWPROP WINDO W 'PROCESS)) and then invokes WINDO W'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 bu er 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 noti ed when it becomes the tty process, or stops being the tty process. For example, Chat (page 20.18) turns o all keyboard interrupt characters while it is the tty process, so that they can be passed transparently to the remote host. 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 (OLDTTYPR OCESS NEWTTYPR OCESS); the new process is made the tty process; nally, the new tty process's TTYENTRYFN is called with two arguments (NEWTTYPR OCESS OLDTTYPR OCESS). Normally the TTYENTRYFN and TTYEXITFN need only their rst 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 con ict 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 rst 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 page

18.32).

18.20.6.2 Handling of Interrupts

At the time that a keyboard interrupt character (page 9.17) 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, ERROR
: (normally control-D and 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-H) 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 always in the tty process.

RUBOUT
: (Initially <del>) 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.

18.20.7 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 nds that another mouse process has sprung up in the meantime. (There is, of course, some computational expense.)

18.20.8 Debugging Processes

(PROCESS.STATUS.WINDOW WHERE) [Function]

Puts up a 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.

The window consists of two menus. The rst is a menu of all the processes at the moment. Commands in the second menu operate on the process selected in the rst menu. The commands are:

BT, BTV, BTV*, BTV!
: Performs a backtrace of the selected process. The rst time, it prompts for a window in which to display the backtrace.

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, 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 info hook can be set using PROCESSPROP.

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).

BREAK
: Enter a break under the selected process. This has the side effect of waking the process with the value returned from the break.

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).

18.20.9 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).

18.21 PROMPTFORWORD

PROMPTFORWORD is a function that reads in a sequence of characters, generally from the keyboard, without involving READ-like syntax. The intent is to mimic the prompted-read used by the Alto Exec when asking for login names, passwords etc. Thus 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, BS, or DEL
: Any of these characters deletes the last character typed and appropriately erases it from the echo stream if it is a displaystream.

Control-W or Control-Q
: Erases all the type-in so far.

Control-R
: Reprints the accumulated string.

?
: Calls up a ''help'' facility. The action taken is de ned by the GENERA TE?LIST.FN argument to PROMPTFORWORD (see below). Normally, this prints a list of possible candidates.

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.

(PROMPTFORWORD PR OMPT.STR CANDID ATE.STR GENERA TE?LIST.FN ECHO.CHANNEL DONTECHOTYPEIN.FL G TIMELIMIT.secs TERMINCHARS.LST KEYBD.CHANNEL OLDSTRING)

[Function]

PROMPTFORWORD has a multiplicity of features, which are speci ed 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 rst of the argument list. The default input and echo are both to the terminal; the terminal table in e ect 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 uses a MONITORLOCK (see page 18.30) so that a second call cannot be started before the rst one nished; primarily this is to limit confusion between multiple processes that might try to access the keyboard at the same time, or print in the prompt window ''at the same time''

PROMPTFORWORD is controlled through the following arguments:

PR OMPT.STR
: If non-NIL, this is coerced to a string and used for prompting; an additional space is output after this string.

CANDID ATE.STR

If non-NIL, this is coerced to a string and offered as initial contents of the input buffer.

GENERA TE?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 GENERA TE?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 file'', 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 ''*''.

TIMELIMIT.secs

If non-NIL, this is the number of seconds (as an integer) that the caller is is willing to wait with no input from KEYBD.CHANNEL (see below); if timeout is reached, then CANDIDATE.WORD is returned, regardless of any other type-in activity.

TERMINCHAR.LST

This is list of ''word terminators''; it defaults to (CHARCODE (EOL ESCAPE LF SPACE TAB)).

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 T, which is a *buffered* keyboard input stream, not suitable for use with PROMPTFORWORD.

OLDSTRING

If non-NIL, this must be a string, which will be destructively used to return the answer.

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* prompting for a candidate, echoing the typed-in characters as ''*''.