35. Miscellaneous Useful Functions

This chapter describes a number of functions that don't logically fit in anywhere else. Most of these functions are not normally used in programs, but are "commands", i.e. things that you type directly at Lisp.

35.1 Documentation

documentation *name doc-type*

Returns the documentation string of *name* in the role *doc-type*. *doc-type* should be a symbol, but only its print-name matters. function as *doc-type* requests the documentation of *name* as a function, variable as *doc-type* requests the documentation of *name* as a variable, and so on.

When *doc-type* is function, *name* can be any function spec, and the documentation string of its function definition is returned. Otherwise, *name* must be a symbol, and *doc-type* may be anything. However, only these values of *doc-type* are standardly used:

variable — Documentation of *name* as a special variable. Such documentation is recorded automatically by defvar, defconst, defconstant, defparameter (page 33).

type — Documentation of *name* as a type for typep. Recorded automatically when a documentation string is given in a deftype form (page 19).

structure — Documentation of *name* as a defstruct type. Recorded automatically by a defstruct for *name* (chapter 20, page 372).

setf — Documentation on what it means to setf a form that starts with *name*. Recorded when there is a documentation string in a defsetf of *name* (page 345).

flavor — Documentation of the flavor named *name*. Put on by the :documentation option in a defflavor for *name* (page 414).

resource — Documentation of the resource named *name*. Put on when there is a documentation string in a defresource of *name* (page 124).

signal — Documentation for *name* as a signal name. Put on when there is a documentation string in a defsignal or defsignal-explicit for *name* (page 714).

Documentation strings for any *doc-type* can be added to *name* by doing (setf (documentation *name doc-type*) *string*).

The command Control-Shift-D in Zmacs and the rubout handler, used within a call to a function, prints the documentation of that function. Control-Shift-V, within a symbol, prints the documentation of that symbol as a variable.

35.2 Hardcopy

The hardcopy functions allow you to specify the printer to use on each call. The default is set up by the site files for your site, but can be overridden for a particular machine in the LMLOCS file or by a user in his INIT file. Any kind of printer can be used, no matter how it is actually driven, if it is hooked into the software properly as described below.

A *printer-type* is a keyword that has appropriate properties; a *printer* is either a printer-type or a list starting with one. The rest of the list can specify *which* printer of that type you want to use (perhaps with a host name or filename).

The printer types defined by the system are:

:dover This printer type is used by itself as a printer, and refers to the Dover at MIT.

:xgp This printer type indicates a printer that is accessed by writing spool files in MIT XGP format. A printer would be specified as a list, (:xgp *filename*), specifying where to write the spool file.

:press-file This printer type is used in a list together with a file name, as in (:press-file "OZ:<RMS>FOO.PRESS"). Something is "printed" on such a printer by being converted to a press file and written under that name.

hardcopy-file *filename* &rest *options*

Print the file *filename* in hard copy on the specified printer or the default printer. *options* is a list of keyword argument names and values. There are only two keywords that are always meaningful: :format and :printer. Everything else is up to the individual printer to interpret. The list here is only a paradigm or suggestion.

:printer The value is the printer to use. The default is the value of si:*default-printer*.

:format The value is a keyword that specifies the format of file to be parsed. The standard possibilities are :text (an ordinary file of text), :xgp (a file of the sort once used by the XGP at MIT), :press (a Xerox-style press file) and :suds-plot (a file produced by the Stanford drawing program). However, each kind of printer may define its own format keywords.

:font

:font-list The value of *font* is the name of a font to print the file in (a string). Alternatively, you can give :font-list and specify a list of such font names, for use if the file contains font-change commands. The interpretation of a font name is dependent on the printer being used. There is no necessary relation to Lisp machine display fonts. However, printers are encouraged to use, by default, fonts that are similar in appearance to the Lisp machine fonts listed in the file's attribute list, if it is a text file.

:heading-font The value is the name of the font for use in page headers, if there are any.

:vsp The value is the extra spacing to use between lines, in over and beyond the height of the fonts.

:page-headings
If the value is non-nil, a heading is added to each page.

:copies The value is the number of copies to print.

:spool If the printer provides optional spooling, this argument says whether to spool (default is nil). Some printers may intrinsically always spool; others may have no way to spool.

set-printer-default-option *printer-type option value*
Sets a default for *option* for printers of type *printer-type*. Any use of the hardcopy functions with a printer of that type and no value specified for *option* will use the value *value*. For example,

```
(set-printer-default-option :dover :spool t)
```

causes output to Dover printers to be spooled unless the :spool option is explicitly specified with value nil.

Currently defaultable options are :font, :font-list, :heading-font, :page-headings, :vsp, :copies, and :spool.

hardcopy-stream *stream* &rest *options*
Like hardcopy-file but uses the text read from *stream* rather than opening a file. The :format option is not allowed (since implementing it requires the ability to open the file with unusual open options).

hardcopy-bit-array *array left top right bottom* &rest *options*
Print all or part of the bit-array *array* on the specified or default printer. *options* is a list of keyword argument names and values; the only standard option is :printer, which specifies the printer to use. The default printer is si:*default-bit-array-printer*, or, if that is nil, si:*default-printer.

left, *top*, *right* and *bottom* specify the subrectangle of the array to be printed. All four numbers measure from the top left corner (which is element 0, 0).

hardcopy-status &optional *printer* (*stream* *standard-output*)
Prints the status of *printer*, or the default printer. This should include if possible such things as whether the printer has paper and what is in the queue.

si:*default-printer* *Variable*
This is the default printer. It is set from the :default-printer site option.

si:*default-bit-array-printer* *Variable*
If non-nil, this is the default printer for printing bit arrays, overriding si:*default-printer*. A separate default is provided for bit arrays since some printers that can print files cannot print bit arrays. This variable is set initially from the :default-bit-array-printer site option.

Defining a printer type:

A printer type is any keyword that has suitable functions on the appropriate properties.

To be used with the function hardcopy-file, the printer type must have a si:print-file property. To be used with hardcopy-stream, the printer type must have a si:print-stream property. hardcopy-bit-array uses the si:print-bit-array property. hardcopy-status uses the si:print-status property. (The hardcopy functions' names are not themselves used simply to avoid using a symbol in the global package as a property name of a symbol that might be in the global package as well).

Each property, to be used, should be a function whose first argument will be the printer and whose remaining arguments will fit the same pattern as those of the hardcopy function the user called. (They will not necessarily be the same arguments, as some additional keywords may be added to the list of keyword arguments; but they will fit the same description.)

For example,

```
(hardcopy-file "foo" :printer '(:press-file "bar.press"))
```

results in the execution of

```
(funcall (get :press-file 'si:print-file)
         '(:press-file "bar.press")
         "foo" :printer '(:press-file "bar.press"))
```

A printer type need not support operations that make no sense on it. For example, there is no si:print-status property on :press-file.

35.3 Metering

The metering system is a way of finding out what parts of your program use up the most time. When you run your program with metering, every function call and return is recorded, together with the time at which it took place. Page faults are also recorded. Afterward, the metering system analyzes the records and tells you how much time was spent executing withain each function. Because the records are stored in the disk partition called METR, there is room for a lot of data.

Before you meter a program, you must enable metering in some or all stack groups. meter:enable is used for this. Then you evaluate one or more forms with metering, perhaps by using meter:test or meter:run. Finally, you use meter:analyze to summarize and print the metering data.

There are two parameters that control whether metering data are recorded. First of all, the variable sys:%meter-microcode-enables contains bits that enable recording of various kinds of events. Secondly, each stack group has a flag that controls whether events are recorded while running in that stack group.

sys:%meter-microcode-enables *Variable*

Enables recording of metering data. Each bit controls recording of one kind of event.

1	This bit enables recording of page faults.
2	This bit enables recording of consing.
4	This bit enables recording of function entry and exit.
8	This bit enables recording of stack group switching.

The value is normally zero, which turns off all recording.

These are the functions used to control which stack groups do metering:

meter:enable &rest *things*

Enables metering in the stack groups specified by *things*. Each thing in *things* may be a stack group, a process (which specifies the process's stack group), or a window (which specifies the window's process's stack group). t is also allowed. It enables metering in all stack groups.

meter:disable &rest *things*

Disables metering in the stack groups specified by *things*. The arguments allowed are the same as for meter:enable. (meter:disable t) turns off (meter:enable t), but does not disable stack groups enabled individually. (meter:disable) disables all stack groups no matter how you specified to enable them.

meter:metered-objects *Variable*

This is a list of all the *things* you have enabled with meter:enable and not disabled.

These are the functions to evaluate forms with metering:

meter:run *forms*

Clears out the metering data and evaluates the *forms* with sys:%meter-microcode-enables bound to 14 octal (record function entry and exit, and stack group switching). Any of the evaluation that takes place in enabled stack groups will record metering data.

meter:test *form* (*enables* #o14)

Clears out the metering data, enables metering for the current stack group only, and evaluates *form* with sys:%meter-microcode-enables bound to *enables*.

This is how you print the results:

meter:analyze &key *analyzer stream file buffer return info* &allow-other-keys

Analyzes the data recorded by metering. *analyzer* is a keyword specifies a kind of analysis. :tree is the default. Another useful alternative is :list-events. Particular analyzers handle other keyword arguments in addition to those listed above.

The output is printed on *stream*, written to a file named *file*, or put in an editor buffer named *buffer* (at most one of these three arguments should be specified). The default is to print on *standard-output*.

Analyzing the metering data involves creating a large intermediate data base. Normally this is created afresh each time meter:analyze is called. If you specify a non-nil value for *return*, the intermediate data structure is returned by meter:analyze, and can be passed in on another call as the *info* argument. This can save time. But you can only do this if you use the same *analyzer* each time, as different analyzers use different termporary data structures.

The default analyzer :tree prints out the amount of run time and real time spent executing each function that was called. The real time includes time spend waiting and time spent writing metering data to disk; for computational tasks, the latter makes the real time less useful than the run time. :tree handles these additional keyword arguments to meter:analyze:

:find-callers The argument for this keyword is a function spec or a list of function specs. A list of who called the specified functions, and how often, is printed instead of the usual output.

:stack-group The argument is a stack group or a list of them; only the activities in those stack groups are printed.

:sort-function The argument is the name of a suitable sorting function that is used to sort the items for the various functions that were called. Sorting functions provided include meter:max-page-faults, meter:max-calls, meter:max-run-time (the default), meter:max-real-time, and meter:max-run-time-per-call.

:summarize The argument is a function spec or a list of function specs; only those functions' statistics are printed.

:inclusive If this is non-nil, the times for each function include the time spent in executing subroutines called from the function.

Note: if a function is called recursively, the time spent in the inner call(s) is counted twice (or more).

The analyzer :list-events prints out one line about each event recorded. The line contains the run time and real time (in microseconds), the running count of page faults, the stack group name, the function that was running, the stack depth, the type of event, and a piece of data. For example:

```
    0    0     0 ZMACS-WINDOWS  METER:TEST   202 CALL SI:EVAL
  115   43     0 ZMACS-WINDOWS  METER:TEST   202 RET  SI:EVAL
  180   87     0 ZMACS-WINDOWS  METER:TEST   202 RET  CATCH

real run   pf  stack-group    function  stack event data
time time                              level type
```

:list-events is often useful with recording of page faults (sys:%meter-microcode-enables set to 1).

meter:reset
Clears out all metering data.

Because metering records pointers to Lisp objects in a disk partition which is not part of the Lisp address space, garbage collection is inhibited (by arresting the gc process) when you turn on metering.

meter:resume-gc-process
Allows garbage collection to continue (if it is already turned on) by unarresting it.

35.4 Poking Around in the Lisp World

who-calls *x* &optional *package* (*inheritors* t) (*inherited* t)
x must be a symbol or a list of symbols. who-calls tries to find all of the functions in the Lisp world that call *x* as a function, use *x* as a variable, or use *x* as a constant. (Constants which are lists containing *x* are not found.) It tries to find all of the functions by searching all of the function cells of all of the symbols in *package* and packages that inherit from *package* (unless *inheritors* is nil) and packages *package* inherits from (unless *inherited* is nil). *package* defaults to the global package, which means that all normal packages are checked.

If who-calls encounters an interpreted function definition, it simply tells you if *x* appears anywhere in the interpreted code. who-calls is smarter about compiled code, since it has been nicely predigested by the compiler. Macros expanded in the compilation of the code can be found because they are recorded in the caller's debugging info alist, even though they are not actually referred to by the compiled code.

If *x* is a list of symbols, who-calls does them all simultaneously, which is faster than doing them one at a time.

who-uses is an obsolete name for who-calls.

The editor has a command, Meta-X List Callers, which is similar to who-calls.

The symbol :unbound-function is treated specially by who-calls. (who-calls :unbound-function) searches all the compiled code for any calls through a symbol that is not currently defined as a function. This is useful for finding errors such as functions you misspelled the names of or forgot to write.

who-calls prints one line of information for each caller it finds. It also returns a list of the names of all the callers.

what-files-call *x* &optional *package* (*inheritors* t) (*inherited* t)
Similar to who-calls but returns a list of the pathnames of all the files that contain functions that who-calls would have printed out. This is useful if you need to recompile and/or edit all of those files.

apropos *substring* &optional *package* &key (*inheritors* t) *inherited dont-print predicate boundp fboundp*

(apropos *substring*) tries to find all symbols whose print-names contain *substring* as a substring. Whenever it finds a symbol, it prints out the symbol's name; if the symbol is defined as a function and/or bound to a value, it tells you so and prints the names of the arguments (if any) to the function.

If *predicate* is non-nil, it should be a function; only symbols on which the function returns non-nil are counted. In addition, *fboundp* non-nil means only symbols with function definitions are considered, and *boundp* non-nil means that only symbols with values are considered.

apropos looks for symbols on *package*, and all packages that use *package* (unless *inheritors* is nil). If *inherited* is non-nil, all packages used by *package* are searched as well. *package* can be a package or a symbol or string naming a package. It can also be a list of packages, symbols and strings; all of the packages thus specified are searched. *package* defaults to a list of all packages except invisible ones.

apropos returns a list of all the symbols it finds. If *dont-print* is non-nil, that is all it does.

sub-apropos *substring starting-list* &key *predicate dont-print*

Finds all symbols in *starting-list* whose names contain *substring*, and that satisfy *predicate*. If *predicate* is nil, the substring is the only condition. The symbols are printed if *dont-print* is nil. A list of the symbols found is returned, in any case.

This function is most useful when applied to the value of *, after apropos has returned a long list.

where-is *pname* &optional *package*

Prints the names of all packages that contain a symbol with the print-name *pname*. If *pname* is a string it gets upper-cased. The package *package* and all packages that inherit from it are searched. *package* can be a package or the name of a package, or a list of packages and names. It defaults to a list of all packages except invisible ones. where-is returns a list of all the symbols it finds.

describe *x*

describe tries to tell you all of the interesting information about any object *x* (except for array contents). describe knows about arrays, symbols, floats, packages, stack groups, closures, and FEFs, and prints out the attributes of each in human-readable form. Sometimes objects found inside *x* are described also; such recursive descriptions are indented appropriately. For instance, describe of a symbol also describes the symbol's value, its definition, and each of its properties. describe of a float (full-size or short) shows you its internal representation in a way that is useful for tracking down roundoff errors and the like.

If *x* is a named-structure, describe invokes the :describe operation to print the description, if that is supported. To understand this, you should read the section on named structures (see page 390). If the :describe operation is not supported, describe

looks on the named-structure symbol for information that might have been left by defstruct; this information would tell it what the symbolic names for the entries in the structure are, and describe knows how to use the names to print out what each field's name and contents is.

describe of an instance always invokes the :describe operation. All flavors support it, since si:vanilla-flavor defines a method for it.

describe always returns its argument, in case you want to do something else to it.

inspect *x*

A window-oriented version of describe. See the window system documentation for details, or try it and type Help.

disassemble *function*

Prints out a human-readable version of the macro-instructions in *function*. *function* should be a FEF, or a function spec whose definition is a FEF. The macro-code instruction set is explained in chapter 31, page 752.

The grindef function (see page 528) may be used to display the definition of a non-compiled function.

room &rest *areas*

Prints a summary of memory usage.

The first line of output tells you the amount of physical memory on the machine, the amount of available virtual memory not yet filled with data (that is, the portion of the available virtual memory that has not yet been allocated to any region of any area), and the amount of wired physical memory (i.e. memory not available for paging).

Following lines tell you how much room is left in some areas. For each area it tells you about, it prints out the name of the area, the number of regions that currently make up the area, the current size of the area in kilowords, and the amount of the area that has been allocated, also in kilowords. If the area cannot grow, the percentage that is free is displayed.

(room) tells you about those areas that are in the list that is the value of the variable room. These are the most interesting ones.

(room *area1 area2*...) tells you about those areas, which can be either the names or the numbers.

(room t) tells you about all the areas.

(room nil) does not tell you about any areas; it only prints the first line of output.

room *Variable*

The value of **room** is a list of area names and/or area numbers, denoting the areas that the function **room** should describe if given no arguments. Its initial value is:

```
(working-storage-area macro-compiled-program)
```

35.5 Utility Programs

ed &optional *x*

ed is the main function for getting into the editor, Zmacs. The commands of Zmacs are very similar to those of Emacs.

(ed) or (ed nil) simply enters the editor, leaving you in the same buffer as the last time you were in the editor. It has the same effect as typing **System E**.

(ed t) puts you in a fresh buffer with a generated name (like BUFFER-4).

(ed *pathname*) edits that file. *pathname* may be an actual pathname or a string.

(ed 'foo) tries hard to edit the definition of the **foo** function. It can find a buffer or file containing the source code for **foo** and position the cursor at the beginning of the code. In general, **foo** can be any function-spec (see section 11.2, page 223).

(ed 'zwei:reload) reinitializes the editor. It forgets about all existing buffers, so use this only as a last resort.

zwei:save-all-files

This function is useful in emergencies in which you have modified material in Zmacs buffers that needs to be saved, but the editor is partially broken. This function does what the editor's **Save All Files** command does, but it stays away from redisplay and other advanced facilities so that it might work if other things are broken.

dired &optional *pathname*

Puts up a window and edits the directory named by *pathname*, which defaults to the last file opened. While editing a directory you may view, edit, compare, hardcopy, and delete the files and subdirectories it contains. While in the directory editor type the **Help** key for further information.

mail &optional *user text call-editor-anyway*

Sends the string *text* as mail to *user*. *user* should also be a string, of the form "*username@hostname*". Multiple recipients separated by commas are also allowed.

If you do not provide two arguments, **mail** puts up an editor window in which you may compose the mail. Type the **End** key to send the mail and return from the **mail** function.

The window is also used if *call-editor-anyway* is non-nil.

bug &optional *topic text call-editor-anyway*

Reports a bug. *topic* is the name of the faulty program (a symbol or a string). It defaults to **lispm** (the Lisp Machine system itself). *text* is a string which contains the information to report. If you do not provide two arguments, or if *call-editor-anyway* is non-**nil**, a window is put up for you to compose the mail.

bug is like **mail** but includes information about the system version and what machine you are on in the text of the message. This information is important to the maintainers of the faulty program; it aids them in reproducing the bug and in determining whether it is one that is already being worked on or has already been fixed.

print-notifications &optional (*from* **0**) *to*

Reprints any notifications that have been received. **from** and **to** are used to restrict which notifications are printed; both count from the most recent notification as number 0. Thus, **(print-notifications 2 8)** prints six notifications after skipping the two most recent.

The difference between notifications and sends is that sends come from other users, while notifications are usually asynchronous messages from the Lisp Machine system itself. However, the default way for the system to inform you about a send is to make a notification! So **print-notifications** *normally* includes all sends as well.

Typing **Terminal 1 N** pops up a window and calls **print-notifications** to print on it.

si:print-disk-error-log

Prints information about the half dozen most recent disk errors (since the last cold boot).

peek &optional *character*

Selects the PEEK utility, which displays various information about the system, periodically updating it. PEEK has several modes, which are entered by typing a single key which is the name of the mode or by clicking on the menu at the top. The initial mode is selected by the argument, *character*. If no argument is given, PEEK starts out by explaining what its modes are.

time *form*

Evaluates *form* and prints the length of time that the evaluation took. The values of *form* are returned.

Note that **time** with no argument is a function to return a time value counting in 60ths of a second; see page 777. This unfortunate collision is a consequence of Common Lisp.

35.6 The Lisp Listen Loop

These functions constitute the Lisp top level read-eval-print loop or *listen loop* and its associated functions.

si:lisp-top-level
This is the first function called in the initial Lisp environment. It calls lisp-reinitialize, clears the screen, and calls si:lisp-top-level1.

lisp-reinitialize
This function does a wide variety of things, such as resetting the values of various global constants and initializing the error system.

si:lisp-top-level1 **terminal-io**
This is the actual listen loop. Within it, *terminal-io* is bound to the argument supplied. This is the stream used for reading and printing if *standard-input* and *standard-output* are synonyms for *terminal-io*, as they normally are.

The listen loop reads a form from *standard-input*, evaluates it, prints the result (with escaping) to *standard-output*, and repeats indefinitely. If several values are returned by the form all of them are printed. Also the values of *, +, -, //, ++, **, +++, *** and *values* are maintained (see below).

break *format-string* &rest *format-args*
Enters a breakpoint loop, which is similar to a Lisp top level loop. *format-string* and the *format-args* are passed to format to print a message.

```
;Breakpoint message; Resume to continue, Abort to quit.
```

and then enters a loop reading, evaluating, and printing forms. A difference between a break loop and the top level loop is that when reading a form, break checks for the following special cases: If the Abort key is typed, control is returned to the previous break or error-handler, or to top-level if there is none. If the Resume key is typed, break returns nil. If the list (return *form*) is typed, break evaluates *form* and returns the result, without ever calling the function return.

Inside the break loop, the streams *standard-output*, *standard-input*, and query-io are bound to be synonymous to *terminal-io*; *terminal-io* itself is not rebound. Several other internal system variables are bound, and you can add your own symbols to be bound by pushing elements onto the value of the variable sys:*break-bindings* (see page 797).

break used to be a special form whose first argument was a string or symbol which was simply printed *without evaluating it*. In order to facilitate conversion, break really still is a special form. If the call appears to use the old conventions, it behaves in the old way, but the compiler issues a warning if it sees such code.

prin1 *Variable*

The value of this variable is normally **nil**. If it is non-**nil**, then the read-eval-print loop uses its value instead of the definition of **prin1** to print the values returned by functions. This hook lets you control how things are printed by all read-eval-print loops—the Lisp top level, the **break** function, and any utility programs that include a read-eval-print loop. It does not affect output from programs that call the **prin1** function or any of its relatives such as **print** and **format**; if you want to do that, read about customizing the printer, on section 23.1, page 513. If you set **prin1** to a new function, remember that the read-eval-print loop expects the function to print the value but not to output a **return** character or any other delimiters.

- *Variable*

While a form is being evaluated by a read-eval-print loop, **-** is bound to the form itself.

+ *Variable*

While a form is being evaluated by a read-eval-print loop, **+** is bound to the previous form that was read by the loop.

***** *Variable*

// *Variable*

While a form is being evaluated by a read-eval-print loop, ***** is set to the result printed the last time through the loop. If there were several values printed (because of a multiple-value return), ***** is bound to the first value. **//** is bound to a list of all the values of the previous form.

If evaluation of a form is aborted, ***** and **//** remain set to the results of the last successfully completed form. If evaluation is successful but printing is aborted, ***** and **//** are already set for the following form.

Note that when using Common Lisp syntax you would type just **/**.

++ *Variable*

+ + holds the previous value of **+**, that is, the form evaluated two interactions ago.

+++ *Variable*

+ + + holds the previous value of **+ +**.

****** *Variable*

//// *Variable*

Hold the previous values of ***** and **//**, that is, the results of the form evaluated two interactions ago. Only forms whose evaluation is successful cause the values of ***** and **//** to move into ****** and **////**.

Note that when using Common Lisp syntax you would type just **//**.

******* *Variable*

////// *Variable*

Hold the previous values of ** and ////, that is, the results of the form evaluated three interactions ago. Note that when using Common Lisp syntax you would type just ///.

values *Variable*

values holds a list of all lists of values produced by evaluation in this Lisp listener. (car *values*) is nearly equivalent to //, (cadr *values*) to ////, and so on. The difference is that an element is pushed on *values* for each form whose evaluation is started. If evaluation is aborted, the element of *values* is nil.

sys:*break-bindings* *Variable*

When break is called, it binds some special variables under control of the list which is the value of sys:*break-bindings*. Each element of the list is a list of two elements: a variable and a form that is evaluated to produce the value to bind it to. The bindings happen sequentially. Users may push things on this list (adding to the front of it), but should not replace the list wholesale since several of the variable bindings on this list are essential to the operation of break.

lisp-crash-list *Variable*

The value of lisp-crash-list is a list of forms. lisp-reinitialize sequentially evaluates these forms, and then sets lisp-crash-list to nil.

In most cases, the *initialization* facility should be used rather than lisp-crash-list. Refer to chapter 33, page 772.

35.7 The Garbage Collector

gc-on

Turns automatic garbage collection on. Garbage collection will happen when and as needed. Automatic garbage collection is off by default.

Since garbage collection works by copying, you are asked for confirmation if there may not be enough space to complete a garbage collection even if it is started immediately.

gc-off

Turns automatic garbage collection off.

gc-on *Variable*

t when garbage collection is on, nil when it is not. You cannot control garbage collection by setting this variable; it exists so you can examine it. In particular, you can tell if the system found it necessary to turn off garbage collection because it was close to running out of virtual memory.

Normally, automatic garbage collection happens in incremental mode; that is, scavenging happens in parallel with computation. Each consing operation scavenges or copies four words per word consed. In addition, scavenging goes on whenever the machine appears idle.

si:inhibit-idle-scavenging-flag *Variable*

If this is non-nil, scavenging is not done during idle time.

If you are running a noninteractive crunching program, the incremental nature of garbage collection may not be helpful. Then you can make garbage collection more efficient by making it a batch process.

si:gc-reclaim-immediately *Variable*

If this variable is non-nil, automatic garbage collection is done as a batch operation: when the garbage collection process decides that the time has come, it copies all the useful data and discards the old address space, running full blast. (It is still possible to use the machine while this is going on, but it is slow.) More specifically, the garbage collection process scavenges and reclaims oldspace immediately right after a flip happens, using all of the machine's physical memory. This variable is only relevant if you have turned on automatic garbage collection with **(gc-on)**.

A batch garbage collection requires less free space than an incremental one. If there is not enough space to complete an incremental garbage collection, you may be able to win by selecting batch garbage collection instead.

si:gc-reclaim-immediately-if-necessary *Variable*

If this variable is non-nil, then automatic garbage collection is done in batch mode if, when the flip is done, there does not seem to be enough space left to do it incrementally. This variable's value is relevant only if **si:gc-reclaim-immediately** is **nil**.

si:gc-flip-ratio *Variable*

This variable tells the garbage collector what fraction of the data it should expect to have to copy, after each flip. It should be a positive number no larger than one. By default, it is one. But if your program is consing considerable amounts of garbage, a value less than one may be safe. The garbage collector uses this variable to figure how much space it will need to copy all the living data, and therefore indirectly how often garbage collection must be done.

si:gc-flip-minimum-ratio *Variable*

This value is used, when non-nil, to control warnings about having too little space to garbage collect. Its value is a positive number no greater than one, just like that of **si:gc-flip-ratio**. The difference between the two is that **si:gc-flip-ratio** controls when garbage collection is *recommended*, whereas **si:gc-flip-minimum-ratio** controls when the system considers the last possible time to do so. If **si:gc-flip-minimum-ratio** is **nil**, **si:gc-flip-ratio** serves both purposes.

Garbage collection is turned off if it appears to be about to run out of memory. You get a notification if this happens. You also get a notification when you are nearly at the point of not having enough space to guarantee garbage collecting successfully.

In addition to turning on automatic garbage collection, you can also manually request one immediate complete collection with the function **si:full-gc**. The usual reason for doing this is to make a band smaller before saving it. **si:full-gc** also resets all temporary areas (see **si:reset-temporary-area**, page 299).

si:full-gc

Performs a complete garbage collection immediately. This does not turn automatic garbage collection on or off; it performs the garbage collection in the process you call it in. A full gc of the standard system takes about 7 minutes, currently.

si:clean-up-static-area *area-number*

This is a more selective way of causing static areas to be garbage collected once. The argument is the area number of a static area; that particular area will be garbage collected the next time a garbage collection is done (more precisely, it will be copied and discarded after the next flip). If you then call si:full-gc, it will happen then.

gc-status

The function gc-status prints information related to garbage collection. When scavenging is in progress, it tells you how the task is progressing. While scavenging is not in progress and oldspace does not exist, it prints information about how soon a new flip will be required.

While a garbage collection is not in progress, the output from gc-status looks like this:

```
Dynamic (new+copy) space 557,417, Old space 0, Static 3,707,242,
Free space 10,453,032, with 10,055,355 needed for garbage collection
 assuming 100% live data (SI:GC-FLIP-RATIO = 1).
If GC is turned on, a flip will happen in 397,677 words.
Scavenging during cons Off, Idle scavenging On,
Automatic garbage collection Off.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

or

```
Dynamic (new+copy) space 561,395, Old space 0, Static 3,707,242,
Free space 10,453,032, with 10,058,670 needed for garbage collection
 assuming 100% live data (SI:GC-FLIP-RATIO = 1).
A flip will happen in 394,362 words.
Scavenging during cons On, Idle scavenging On,
Automatic garbage collection On.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

The "dynamic space" figure is the amount of garbage collectable space and the "static" figure is the amount of static space used. There is no old space since an old space only exists during garbage collection.

The amount of space needed for garbage collection represents an estimate of how much space user programs will use up while scavenging is in progress. It includes a certain amount of padding. The difference between the free space and that amount is how much consing you can do before a garbage collection will begin (if automatic garbage collection is on).

The amount needed for a garbage collection depends on the value of si:*gc-reclaim-immediately*; more if it is nil.

While a garbage collection is in progress, the output looks like this:

```
Incremental garbage collection now in progress.
Dynamic (new+copy) space 45,137, Old space 972,514, Static 3,707,498,
Between 3,701,440 and 4,629,998 words of scavenging left to do.
Free space 9,289,795 (of which 928,558 might be needed for copying).
Ratio scavenging work/free space = 0.55.
Scavenging during cons On, Idle scavenging On,
Automatic garbage collection On.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

Notice that most of the dynamic space has become old space and new space is small. Not much has been copied since the flip took place. The maximum and minimum estimates for the amount of scavenging are based on different limits for how much of old space may need to be copied; as scavenging progresses, the maximum decreases steadily, but the minimum may increase. The free space is smaller now, but it will get larger when scavenging is finished and old space is freed up. (The total amounts are not the same now because unused parts of regions may not be included in any of the figures.)

si:set-scavenger-ws *number-of-pages*

Incremental scavenging is restricted to a fixed amount of physical memory to reduce its interference with your other activities.

This function specifies the number of pages of memory that incremental garbage collection can use. 256 is a good value for a 256k machine. If the garbage collector gets very poor paging performance, use of this function may fix it.

35.8 Logging In

Logging in tells the Lisp Machine who you are, so that other users can see who is logged in, you can receive messages, and your INIT file can be run. An INIT file is a Lisp program which gets loaded when you log in; it can be used to set up a personalized environment.

When you log out, it should be possible to undo any personalizations you have made so that they do not affect the next user of the machine. Therefore, anything done by an INIT file should be undoable. In order to do this, for every form in the INIT file, a Lisp form to undo its effects should be added to the list that is the value of **logout-list**. The **login-forms** construct helps make this easy; see below.

user-id *Variable*

The value of user-id is either the name of the logged in user, as a string, or else an empty string if there is no user logged in. It appears in the who-line.

logout-list *Variable*

The value of **logout-list** is a list of forms to be evaluated when the user logs out.

login *name* &optional *host inhibit-init-file*

Sets your name (the variable **user-id**) to *name* and logs in a file server on *host*. *host* also becomes your default file host. The default value of *host* depends on which Lisp Machine you use using; it is called the associated machine (see page 815). **login** also runs the :**login** initialization list (see page 773).

If *host* requires passwords for logging in you are asked for a password. Adding an asterisk at the front of your password enables any special capabilities you may be authorized to use, by calling **fs:enable-capabilities** (page 609).

Unless *inhibit-init-file* is specified as non-**nil**, **login** loads your init file if it exists. On ITS, your init file is *name* **LISPM** on your home directory. On TOPS-20 your init file is **LISPM.INIT** on your directory. On VMS, it is **LISPM.INI**. On Unix, it is **lispm.init**.

If anyone is logged into the machine already, **login** logs him out before logging in *name*. (See **logout**.) Init files should be written using the **login-forms** construct so that **logout** can undo them. Usually, however, you cold-boot the machine before logging in, to remove any traces of the previous user. **login** returns **t**.

log1 &rest *options*

Like **login** but the arguments are specified differently. *options* is a list of keywords and values; the keywords :**host** and :**init** specify the host to log in on and whether to load the init file if any. Any other keywords are also allowed. **log1** itself ignores them, but the init file can act on them. The purpose of **log1**, as opposed to **login**, is to enable you to specify other keywords for your init file's sake.

si:user-init-options *Variable*

During the execution of the user's init file, inside **log1**, this variable contains the arguments given to **log1**. Options not meaningful to **log1** itself can be specified, so that the init file can find them here and act on them.

logout

First, **logout** evaluates the forms on **logout-list**. Then it sets **user-id** to an empty string and **logout-list** to **nil**. Then it runs the :**logout** initialization list (see page 773), and returns **t**.

login-forms *undoable-forms...* *Macro*

The body of a **login-forms** is composed of forms to be evaluated, whose effects are to be undone if you log out. For example,

```
(login-forms
  (setq fs:*defaults-are-per-host* t))
```

would set the variable immediately but arrange for its previous value to be restored if you log out.

login-forms is not an AI program; it must be told how to undo each function that will be used immediately inside it. This is done by giving the function name (such as **setq**) a :**undo-function** property which is a function that takes a form as an argument and returns a form to undo the original form. For **setq**, this is done as follows:

```
(defun (setq :undo-function) (form &aux results)
  (do ((l (cdr form) (cddr l)))
      ((null l))
    (cond ((boundp (car l))
           (push '(setq ,(car l) ',(symeval (car l))) results))
          (t (push '(makunbound ',(car l)) results))))
  '(progn . ,results))
```

Undo functions are standardly provided for the functions setq, pkg-goto-globally, setq-globally, add-initialization, deff, defun, defsubst, macro, advise and zwei:set-comtab. Constructs which macroexpand into uses of those functions are also supported.

Note that setting *read-base* and *print-base* should be done with setq-globally rather than setq, since those variables are likely to be bound by the load function while the init file is executed.

login-setq {*variable value*}... *Macro*

login-setq is like setq except that it puts a setq form on logout-list to set the variables to their previous values. login-setq is obsolete; use login-forms around a setq instead.

login-eval *x*

login-eval is used for functions that are "meant to be called" from INIT files, such as zwei:set-comtab-return-undo, which conveniently return a form to undo what they did. login-eval pushes the result of the form *x* onto logout-list. It is obsolete now because login-forms is a cleaner interface.

si:undoable-forms-1 *undo-list-name forms* &optional *complaint-string*

This is what login-forms uses. *forms* is a list of forms; they are evaluated and forms for undoing their effects are pushed onto the value of the symbol *undo-list-name*. If an element of *forms* has no known way to be undone, a message is printed using the string *complaint-string*. For login-forms, the string supplied is "at logout".

35.9 Dribble Files

dribble &optional *filename*

With an argument, dribble opens *filename* as a 'dribble file' (also known as a 'wallpaper file') and then enters a Lisp listen loop in which *standard-input* and *standard-output* are rebound to direct all the output and echoing they do to the file as well as to the terminal.

Dribble output can be sent to an editor buffer by using a suitable pathname; see section 24.7.6, page 575.

Calling dribble with no arguments terminates dribbling; it throws to the original call to dribble, which closes the file and returns.

dribble-all &optional *filename*

Like dribble except that all input and output goes to the dribble file, including break loops, queries, warnings and sessions in the debugger. This works by binding *terminal-io* instead of *standard-output* and *standard-input*.

35.10 Version Information

Common Lisp defines several standard ways of inquiring about the identity and capabilities of the Lisp system you are using.

features *Variable*

A list of atoms which describe the software and hardware features of the Lisp implementation. By default, this is

```
(:loop :defstruct :lispm :cadr :mit :chaos :sort :fasload :string
 :newio :roman :trace :grindef :grind :common)
```

Most important is the symbol :lispm; this indicates that the program is executing on the Lisp Machine. :cadr indicates the type of hardware, :mit which version of the Lisp Machine operating system, and :chaos that the Chaosnet protocol is available. :common indicates that Common Lisp is supported.

Most of the other elements are for Maclisp compatibility. Common Lisp defines the variable *features* but does not define what should appear in the list. The order of elements in the list has no significance. Membership checks should use string-equal so that packages are not significant

The #+ and #- read constructs (page 525) check for the presence of an element in this list. Thus, #+lispm when read by a Lisp Machine causes the following expression to be significant, because :lispm is present in the features list.

The remaining standard means of inquiry are specified by Common Lisp to be functions rather than variables, for reasons that seem poorly thought out.

lisp-implementation-type

Returns a string saying what kind of Lisp implementation you are using. On the Lisp Machine it is always "Zetalisp".

lisp-implementation-version

Returns a string saying the version numbers of the Lisp implementation. On the Lisp Machine it looks something like

"System 98.3, CADR 3.0, ZMAIL 52.2".

machine-type

Returns a string describing the kind of hardware in use. It is "CADR" or "LAMBDA".

machine-version
Returns a string describing the kind of hardware and microcode version. It starts with the value of machine-type. It might be "CADR Microcode 309".

machine-instance
Returns a string giving the name of this machine. Do not be confused; the value is a string, not an instance. Example: "CADR-18".

software-type
Returns a string describing the type of operating system software that Lisp is working with. On the Lisp Machine, it is always "Zetalisp", since the Lisp Machine Lisp software is the operating system.

software-version
Returns a string describing the version numbers of the operating system software in use. This is the same as lisp-implementation-version on the Lisp Machine since the same software is being described.

short-site-name
Returns a string giving briefly the name of the site you are at. A site is an institution which has a group of Lisp Machines. The string you get is the value of the :short-site-name site option as given in SYS: SITE; SITE LISP. See section 35.12, page 810 for more information. Example: "MIT AI Lab".

long-site-name
Returns a string giving a verbose name for the site you are at. This string is specified by the site option :long-site-name. Example: "Massachusetts Institute of Technology, Artificial Intelligence Laboratory".

35.11 Booting and Disk Partitions

A Lisp Machine disk is divided into several named *partitions* (also called *bands* sometimes). Partitions can be used for many things. Every disk has a partition named PAGE, which is used to implement the virtual memory of the Lisp Machine. When you run Lisp, this is where the Lisp world actually resides. There are also partitions that hold saved images of the Lisp Machine microcode, conventionally named MCR*n* (where *n* is a digit), and partitions that hold saved images of Lisp worlds, conventionally named LOD*n*. A saved image of a Lisp world is also called a *virtual memory load* or *system load*. The microcode and system load are stored separately so that the microcode can be changed without going through the time-consuming process of generating a new system load.

The directory of partitions is in a special block on the disk called the label. The label names one of the partitions as the current microcode and one as the current system load. When you cold-boot, the contents of the current microcode band are loaded into the microcode memory, and then the contents of the current saved image of the Lisp world is copied into the PAGE partition. Then Lisp starts running. When you warm-boot, the contents of the current microcode band are loaded, but Lisp starts running using the data already in the PAGE partition.

For each partition, the directory of partitions contains a brief textual description of the contents of the partition. For microcode partitions, a typical description might be "UCADR 310"; this means that version 310 of the microcode is in the partition. For saved Lisp images, it is a little more complicated. Ideally, the description would say which versions of which systems are loaded into the band. Unfortunately, there isn't enough room for that in most cases. A typical description is "99.4 Daed 5.1", meaning that this band contains version 99.4 of System and version 5.1 of Daedalus. The description is created when a Lisp world is saved away by disk-save (see below).

35.11.1 Manipulating the Label

print-disk-label &optional (*unit* 0) (*stream* *standard-output*)

Prints a description of the label of the disk specified by *unit* onto *stream*. The description starts with the name of the disk pack, various information about the disk that is generally uninteresting, and the names of the two current load partitions (microcode and saved Lisp image). This is followed by one line of description for each partition. Each one has a name, disk address, size, and textual comment. The current microcode partition and the current system load partition are marked with asterisks, each at the beginning of the line.

unit may be the unit number of the disk (most Lisp machines just have one unit, number 0), or the host name of another Lisp Machine on the Chaosnet, as a string (in which case the label of unit 0 on that machine is printed, and the user of that machine is notified that you are looking at his label), or, for CADRs only, the string "CC" (which prints the label of unit 0 of the machine connected to this machine's debugging hardware).

Use of "CC" as the *unit* is the way to examine or fix up the label of a machine which cannot work because of problems with the label. On a Lambda, this must be done through the SDU.

set-current-band *partition-name* &optional (*unit* 0)

Sets the current saved Lisp image partition to be *partition-name*. If *partition-name* is a number, the name LOD*n* is used.

unit can be a disk drive number, the host name of another Lisp Machine, or the string "CC". See the comments under print-disk-label, above.

If the partition you specify goes with a version of microcode different from the one that is current, this function offers to select the an appropriate microcode partition as well. Normally you should answer Y.

set-current-microload *partition-name* &optional (*unit* 0)

Sets the current microcode partition to be *partition-name*. If *partition-name* is a number, the name MCR*n* is used.

unit can be a disk drive number, the host name of another Lisp Machine, or the string "CC". See the comments under print-disk-label, above.

si:current-band &optional (*unit* 0)
si:current-microload &optional (*unit* 0)
Return, respectively, the name of the current band and the current microload on the specified unit.

When using the functions to set the current load partitions, be extra sure that you are specifying the correct partition. Having done it, cold-booting the machine will reload from those partitions. Some versions of the microcode will not work with some versions of the Lisp system, and if you set the two current partitions incompatibly, cold-booting the machine will fail. To fix this, on a CADR, use another CADR's debugging hardware, running print-disk-label and set-current-band on the other CADR and giving "CC" as the *unit* argument. On a Lambda, this is done via the SDU.

si:edit-disk-label *unit* &optional *init-p*
Runs an interactive label editor on the specified unit. This editor allows you to change any field in the label. The Help key documents the commands. You have to be an expert to need this and to understand what it does, so the commands are not documented here. Ask someone if you need help. You can screw yourself very badly with this function.

disk-restore &optional *partition*
Allows booting from a band other than the current one. *partition* may be the name or the number of a disk partition containing a virtual-memory load, or nil or omitted, meaning to use the current partition. The specified partition is copied into the paging area of the disk and then started.

Although you can use this to boot a different Lisp image than the installed one, this does not provide a way to boot a different microcode image. disk-restore brings up the new band with the currently running microcode.

disk-restore asks the user for confirmation before doing it.

describe-partition *partition* &optional *unit*
Tells you various useful things about a partition; including where on disk the partition begins, and how long it is.

If you specify a saved Lisp system partition, such as LOD3, it also tells you important information about the contents of the partition: the microcode version which the partition goes with, the size of the data in the partition and the highest virtual address used. The size of the partition tells how large a partition you need to make a copy of this one, and the highest virtual address used (which is measured in units of disk blocks) tells you how large a PAGE partition you need in order to run this partition.

35.11.2 Updating Software

Of all the procedures described in this section, the most common one is to take a partition containing a Lisp image, update it to have all the latest patches (see section 28.8, page 672), and save it away in a disk partition. The function load-and-save-patches does it all conveniently for you.

load-and-save-patches
Loads patches and saves a band, with a simple user interface. Run this function immediately after cold booting, without logging in first; it logs in automatically as LISPM (or whatever is specified in the site files). The first thing it does is print the list of disk partitions and ask you which one to save in. Answer LOD*n*, using the name of a partition from the printed list. You must then confirm. Then the patches are loaded and the resulting world is saved with no further user interaction, as long as no problem arises.

It is convenient to use this function just before you depart, allowing it to finish unattended.

If you wish to do something other than loading all and only the latest patches, you must perform the steps by hand. Start by cold-booting the machine, to get a fresh, empty system. Next, you must log in as something whose INIT file does not affect the Lisp world noticably (so that when you save away the Lisp image, the side-effects of the INIT file won't get saved too); on MIT-OZ, for example, you can log in as LISPM with password LISPM. Now you can load in any new software you want; usually you should also do (load-patches) for good measure. You may also want to call si:set-system-status to change the release status of the system.

When you're done loading everything, do (print-disk-label) to find a band in which to save your new Lisp world. It is best not to reuse the current band, since if something goes wrong during the saving of the partition, while you have written, say, half of the band that is current, it may be impossible to cold-boot the machine. Once you have found the partition, you use the disk-save function to save everything into that partition.

disk-save *partition-name* &optional *no-query incremental*
Saves the current Lisp world in the designated partition. *partition-name* may be a partition name (a string), or it may be a number in which case the name LOD*n* is used.

The user is first asked for yes-or-no confirmation that he really wants to reuse the named partition. A non-nil value for *no-query* prevents this question. This is only for callers that have already asked.

Next it is necessary to figure out what to put into the textual description of the band, for the disk label. This starts with the brief version of si:system-version-info (see page 674). Then comes a string of additional information; if *no-query* is nil, the user is offered the chance to provide a new string. The current value of this string is returned by si:system-version-info and printed by booting. The version info and the string both go in the comment field of the disk label for this band. If they don't together fit into the fixed size available, the user is asked to retype the whole thing (the version info as well as your comment) in a compressed form that does fit.

The Lisp environment is then saved away into the designated partition, and then the equivalent of a cold-boot from that partition is done.

Once the patched system has been successfully saved and the system comes back up, you can make it current with **set-current-band**.

When you do a **disk-save**, it may tell you that the band you wish to save in is not big enough to hold all the data in your current world. It may be possible for you to reduce the size of the data so that it will fit in that band, by garbage collecting. Simply do **(si:full-gc)**.

Try to avoid saving patched systems after running the editor or the compiler. This works, but it makes the saved system a lot bigger. In order to produce a clean saved environment, you should try to do as little as possible between the time you cold-boot and the time you save the partition.

si:login-history *Variable*

The value of **si:login-history** is a list of entries, one for each person who has logged into this world since it was created. This makes it possible to tell who **disk-saved** a band with something broken in it. Each entry is a list of the user ID, the host logged into, the Lisp Machine on which the world was being executed, and the date and time.

35.11.3 Saving Personal Software

If you have a large application system which takes a while to load, you may wish to save a band containing it.

To do this, boot a fresh band, log in without running your init file, do **make-system** to load the application system, and then invoke **disk-save**. When **disk-save** asks for an additional comment, give your name or the name of the application system you loaded, and a date. This will tell other people who to ask whether the band is still in use if they would like to save other things.

You can greatly reduce the amount of disk space needed for the saved band by making it an *incremental* band; that is, a band which contains the differences between the Lisp world you want to save and the system band you originally loaded. Since all the pages of the system which your application program did not change do not have to be saved, an incremental band is generally much smaller—perhaps by a factor of ten.

To make an incremental band, give a non-nil third argument to **disk-save**, as in

```
(disk-save "lod4" nil t)
```

Figuring out which pages need to be saved in the incremental band takes a couple of extra minutes.

You can restore the incremental band with **disk-restore** or boot it like any other band. This works by first booting the original band and then copying in the differences that the incremental band records. It takes only a little longer than booting the original system band.

The original band to which an incremental band refers must be a complete load. When you update a standard system band (loading patches, for instance) you should always make a complete load, so that the previous system band is not needed for the new one to function.

The incremental band records the partition name of the original system band. That original band must still exist, with the same contents, in order for the incremental band to work properly. The incremental band contains some error check data which is used to verify this. The error checking is done by the microcode when the incremental band is booted, but it is also done by set-current-band, so that you are not permitted to select an incremental band if it is not going to work.

When using incremental bands, it is important to preserve the system bands that they depend on. Therefore, system bands should not be updated too frequently. describe-partition on an incremental band says which full band it depends on; you can use this to determine which bands should be kept for the sake of incremental bands that depend on them.

In order to realize the maximum savings in disk space possible because of incremental bands, you must make the partition you saved in smaller once the save is finished and you know how much space was actually used. This is done with si:edit-disk-label. The excess space at the end of the partition can be used to make another partition which is used for the next incremental band saved. Eventually when some of the incremental bands are no longer needed the rest must be shuffled so that the free space can be put together into larger partitions. This can be done with si:copy-disk-partition.

An easier technique is to divide a couple of the initial partitions into several equal-sized partitions of about 4000 pages, and use these for all incremental saving. You can easily provide room for 12 incremental bands this way in addition to a few system bands and file system.

You must not do a garbage collection to reduce the size of the world before you make an an incremental band. This is because garbage collection alters so many pages that an incremental band would be as big as a complete band.

35.11.4 Copying Bands

The normal way to install new software on a machine is to copy the microcode and world load bands from another machine.

The first step is to find a machine that is not in use and has the desired system. Let us call this the source machine. The machine where the new system is to be installed is the target machine. You can use finger to see which machines are free, and use print-disk-label with an argument to examine the label of that machine's disk and see if it has the system you want.

Then you should do a (print-disk-label) to find suitable partitions to copy them into. It is advisable not to copy them into the selected partitions; if you did that, and the machine crashed in the middle, you would be unable to boot it.

Before copying a band from another machine, double-check the partition names by printing the labels of both machines, and make sure no one is using the other machine. Also double-check with describe-partition that the world load and microcode go together. Then use this

function:

si:receive-band *source-host source-band target-band* &optional *subset-start subset-size*
Copies the partition on *source-host*'s partition named *source-band* onto the local machine's partition named *target-band*. This takes about ten minutes. It types out the size of the partition in pages, and types a number every 100 pages telling how far it has gotten. It displays an entry in the who line on the remote machine saying what's going on.

The *subset-start* and *subset-size* arguments can be used to transfer only part of a partition. They are measured in blocks. The default for the first is zero, and the default for the second is to continue to the end of the data in the band. These arguments are useful for restarting a transfer that was aborted due to network problems or a crash, based on the count of hundreds of blocks that was printed out before the crash.

To go the other direction, use si:transmit-band.

si:transmit-band *source-band target-host target-band* &optional *subset-start subset-size*
This is just like si:receive-band, except you use it on the source machine instead of the target machine. It copies the local machine's partition named *source-band* onto *target-machine*'s partition named *target-band*.

It is preferable to use si:receive-band so that you are present at the machine being written on.

After transferring the band, it is good practice to make sure that it really was copied successfully by comparing the original and the copy. All of the known reasons for errors during band transfer have (of course) been corrected, but peace of mind is valuable. If the copy was not perfectly faithful, you might not find out about it until a long time later, when you use whatever part of the system that had not been copied properly.

si:compare-band *source-host source-band target-band* &optional *subset-start subset-size*
This is like si:receive-band, except that it does not change anything. It compares the two bands and complains about any differences.

Having gotten the current microcode load and system load copied into partitions on your machine, you can make them current for booting using set-current-band.

35.12 Site Options and Host Table

The Lisp Machine system has options that are set at each site. These include the network addresses of other hosts, which hosts have file servers, which host to find the system source files and patch files on, where to send bug reports, what timezone the site is located in, and many other things.

The per-site information is defined by three files: SYS: SITE; SITE LISP, SYS: SITE; LMLOCS LISP, and SYS: CHAOS; HOSTS TXT.

SYS: CHAOS; HOSTS TXT is the network host table. It gives the names and addresses of all hosts that are to be known to the Lisp Machine for any purposes. It also says what type of machine the host is, and what operating system runs on it.

SYS: SITE; LMLOCS LISP specifies various information about the Lisp Machines at your site, including its name, where it is physically located, and what the default machine for logging in should be.

SYS: SITE; SITE LISP specifies all other site-specific information. Primarily, this is contained in a call to the special form **defsite**.

defsite *site-name* (*site-option value*)... *Macro*

This special form defines the values of site-specific options, and also gives the name of the site. Each *site-option* is a symbol, normally in the keyword package, which is the name of some site option. *value* is the value for that option; it is evaluated. Here is a list of standardly defined site options:

:sys-host The value is a string, the name of the host on which the system source files are stored. This host becomes the translation of logical host SYS.

:sys-host-translation-alist

The value is an alist mapping host names into translation-list variables. Each translation list variable's value should be an alist suitable for being the third argument to fs:add-logical-pathname-host (see page 574). The car of an element may be nil instead of a host name; then this element applies to all hosts not mentioned.

The normal place to find the system sources is on the host specified by the :sys-host keyword, in the directories specified by the translation list variable found by looking that host up in the value of the :sys-host-translation-alist keyword. If you specify a different host as the system host with si:set-sys-host, that host is also looked up in this alist to find out what directories to use there.

Here is what is used at MIT:

```
(defsite :mit
  ...
  (:sys-host-translation-alist
    '(("AI" . its-sys-pathname-translations)
      ("OZ" . oz-sys-pathname-translations)
      ("FS" . its-sys-pathname-translations)
      ("LM" . its-sys-pathname-translations)
      (nil . its-sys-pathname-translations)))
  ...)
```

```
(defconst oz-sys-pathname-translations
  '(("CC;" "<L.CC>")
    ("CHAOS;" "<L.CHAOS>")
    ("DEMO;" "<L.DEMO>")
    ...
    ("SITE;" "<L.SITE>")
    ("SYS;" "<L.SYS>")
    ("SYS2;" "<L.SYS2>")
    ...
    ("ZMAIL;" "<L.ZMAIL>")
    ("ZWEI;" "<L.ZWEI>")
    ))
```

:sys-login-name
:sys-login-password
These specify the username and password to use to log in automatically to read system patch files, microcode symbol tables and error tables. The values should be strings.

:chaos nil if the site has no Chaosnet; otherwise, a string, the name of the Chaosnet that the site is on. Names for Chaosnets will eventually be used to permit communication between Chaosnets, probably through special gateway servers. Except when multiple sites are on a single Chaosnet, normally the Chaosnet name should be the same as the site name (but as a string, not a symbol).

:standalone The value should be t for a Lisp Machine that is operated without a network connection. This causes the Lisp Machine to not to try to use the Chaosnet for getting the time. On the Lambda, the time will obtained from the SDU's clock. On the CADR, the time will be obtained from the user.

:default-associated-machine
This should be a string which is the name of a host to use as the associated host for any Lisp Machine not mentioned in the LMLOCS file.

:usual-lm-name-prefix
This should be a string which is the typical beginning of host names of Lisp Machines at your site. At MIT, it is "CADR-".

:chaos-file-server-hosts
This should be a list of names of hosts that have file servers, including Lisp Machines which other Lisp Machines should know about.

:lmfile-server-hosts
This should be a list of names of Lisp Machines that provide servers for the LMFILE file system. The entry for such a machine should be one of the nicknames of that machine. By virtue of its presence in this list, it becomes the name by which the LMFILE file system there can be accessed remotely.

:chaos-time-server-hosts
This should be a list of names of hosts that support TIME servers. These

are hosts that the Lisp Machine can ask the time of day from when you boot.

:chaos-host-table-server-hosts
This should be a list of names of hosts that support host-table servers, which can be used to inquire about hosts on networks that the Lisp Machine does not know about in its own host table.

:chaos-mail-server-hosts
This should be a list of names of hosts that support mail servers which are capable of forwarding mail to any known host.

:timezone This should be a number, the number of hours earlier than GMT of standard time in the timezone where this site is located.

:host-for-bug-reports
This should be a string, the name of the host at which bug-report mailboxes are located.

:local-mail-hosts
This should be a list of names of hosts that ZMail should consider "local" and omit from its summary display.

:spell-server-hosts
This should be a list of hosts that have spelling corrector servers.

:comsat This should be t if mail can be sent through the COMSAT mail demon. This is true only at MIT.

:default-mail-mode
This should be the default mode for use in sending mail. The options are :file (use COMSAT), :chaos (use one of the :chaos-mail-server-hosts), or :chaos-direct (like :chaos, but go direct to the host that the mail is addressed to whenever possible).

:gmsgs This should be t if GMSGS servers are available.

:arpa-gateways
This should be a list of names of hosts that can be used as gateways to the Arpanet. These hosts must provide a suitable Chaosnet server which will make Arpanet connections. It should be nil if your site does not have an Arpanet connection.

:arpa-contact-name
If you have Arpanet gateways, this is the Chaosnet contact name to use. Nowadays, it should be "TCP".

:dover This should be t if your site has a Dover printer.

:default-printer
This should be a keyword which describes the default printer for hardcopy commands and functions to use. Possible values include :dover, nil, or any other printer type that you define (see section 35.2, page 785).

:default-bit-array-printer
Like :default-printer, but this is the default for only hardcopy-bit-array

to use.

:esc-f-arg-alist
This says what various numeric arguments to the Terminal F command mean. It is a list of elements, one for each possible argument. The car of an element is either a number or nil (which applies to Terminal F with no argument). The cdr is either :login (finger the login host), :lisp-machines (finger all Lisp Machines at this site), :read (read some hosts from the keyboard), or a list of host names.

:verify-lm-dumps
If the value is t, Lisp Machine file system dump tapes are verified.

Other site options are allowed, and your own software can look for them.

35.12.1 Updating Site Information

To update the site files, you must first recompile the sources. Do this by

```
(make-system 'site 'compile)
```

This also loads the site files.

To just load the site files, assuming they are compiled, do

```
(make-system 'site)
```

load-patches does that automatically.

You should never load any site file directly. All the files must be loaded in the proper fashion and sequence, or the machine may stop working.

35.12.2 Accessing Site Options

Programs examine the site options using these variables and functions:

site-name *Variable*
The value of this variable is the name of the site you are running at, as defined in the defsite in the SITE file. You can use this in run-time conditionals for various sites.

get-site-option *keyword*
Returns the value of the site option *keyword*. The value is nil if *keyword* is not mentioned in the SITE file.

define-site-variable *variable keyword* [*documentation*] *Macro*
Defines a variable named *variable* whose value is always the same as that of the site option *keyword*. When new site files are loaded, the variable's value is updated. *documentation* is the variable's documentation string, as in defvar.

define-site-host-list *variable keyword* [*documentation*] *Macro*

Defines a variable named *variable* whose value is a list of host objects specified by the site option *keyword*. The value actually specified in the SITE file should be a list of host names. When new site files are loaded, the variable's value is updated. *documentation* is the variable's documentation string, as in **defvar**.

35.12.3 The LMLOCS File

The LMLOCS file contains an entry for each Lisp Machine at your site, and tells the system whatever it needs to know about the particular machine it is running on. It contains one form, a defconst for the variable machine-location-alist. The value should have an element for each Lisp Machine, of this form:

```
("MIT-LISPM-1"  "Lisp Machine One"
 "907 [Son of CONS] CADR1's Room x6765"
 (MIT-NE43 9) "OZ" ((:default-printer :dover)))
```

The general pattern is

(*host-full-name pretty-name*
location-string
(*building floor*) *associated-machine site-options*)

The *host-full-name* is the same as in the host table.

The *pretty-name* is simply for printing out for users on certain occasions.

The *location-string* should say where to find the machine's console, preferably with a telephone number. This is for the FINGER server to provide to other hosts.

The *building* and *floor* are a somewhat machine-understandable version of the location.

The *associated-machine* is the default file server host name for login on this Lisp Machine.

site-options is a list of site options, just like what goes in the defsite. These site options apply only to the particular machine, overriding what is present in the SITE file. In our example, the site option :default-printer is specified as being :dover, on this machine only.

si:associated-machine *Variable*

The host object for the associated machine of this Lisp Machine.