32. 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.

32.1 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 standard/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.

: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.

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 themselves are not 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"))
```

will result 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.

32.2 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 will analyze the records and tell 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.
10	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* 14)
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 &optional *analyzer stream file buffer return info*
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:analyzeis 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) will be 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 turned off (by arresting the gc process) when you turn on metering.

meter:resume-gc-process

Allows garbage collection to continue (if it is on) by unarresting it.

32.3 Poking Around in the Lisp World

who-calls *x* &optional *package*
who-uses *x* &optional *package*

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. (It won't find things that use constants that contain *x*, such as a list one of whose elements is *x*; it will only find it if *x* itself is used as a constant.) It tries to find all of the functions by searching all of the function cells of all of the symbols on *package* and *package*'s descendants. *package* defaults to the global package, and so normally all 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.

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) will search 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*

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* &key &optional *package* (*inferiors* t) *superiors dont-print predicate*

(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.

apropos looks for symbols on *package*, and *package*'s decendants (unless *inferiors* is nil). If *superiors* is t, the superpackages of the specified package are searched as well. *package* defaults to the global package, so normally all packages are searched.

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 &optional *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 its sub-packages are searched; *package* defaults to the global package, which causes all packages to be searched. 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, flonums, packages, stack groups, closures, and FEFs, and prints out the attributes of each in human-readable form. Sometimes it will describe something that it finds inside something else; such recursive descriptions are indented appropriately. For instance, describe of a symbol will tell you about the symbol's value, its definition, and each of its properties. describe of a flonum (regular or small) will show 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 handles it specially. To understand this, you should read the section on named structures (see page 312). First it gets the named-structure symbol, and sees whether its function knows about the :describe operation. If the operation is known, it applies the function to two arguments: the symbol :describe, and the named-structure itself. Otherwise, it 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 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*

function should be a FEF, or a symbol that is defined as a FEF. This prints out a human-readable version of the macro-instructions in *function*. The macro-code instruction set is explained in chapter 28, page 602.

The grindef function (see page 426) may be used to display the definition of a non-compiled function.

room &rest *areas*

room 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). Then it tells 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 header. This is useful if you just want to know how much memory is on the machine or how much virtual memory is available.

room *Variable*

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

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

32.4 Utility Programs

ed &optional *x*

ed is the main function for getting into the editor, Zwei. Zwei is not yet documented in this manual, but the commands are very similar to Emacs.

(ed) or (ed nil) simply enters the editor, leaving you in the same buffer as the last time you were in the editor.

(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 will 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 10.2, page 154).

(ed 'zwei:reload) reinitializes the editor. It will forget 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 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 will be 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
Reprints any notifications that have been received. 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** will *normally* include all sends as well.

si:print-disk-error-log
Prints information about the half dozen most recent disk errors (since the last cold boot).

peek &optional *character*
peek is similar to the ITS program of the same name. It displays various information about the system, periodically updating it. Like ITS PEEK, it has several modes, which are entered by typing a single key which is the name of the mode. The initial mode is selected by the argument, *character*. If no argument is given, **peek** starts out by explaining what its modes are.

32.5 The Lisp Top Level

These functions constitute the Lisp top level 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 top level loop. It reads a form from standard-input, evaluates it, prints the result (with slashification) to standard-output, and repeats indefinitely. If several values are returned by the form all of them will be printed. Also the values of *, +, -, //, ++, **, +++, and *** are maintained (see below).

break [*tag*] [*conditional-form*] *Special Form*

break is used to enter a breakpoint loop, which is similar to a Lisp top level loop. (break *tag*) will always enter the loop; (break *tag conditional-form*) will evaluate *conditional-form* and only enter the break loop if it returns non-nil. If the break loop is entered, break prints out

```
;Breakpoint tag; 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 symbol ◊p is typed, break returns nil. If the list (return *form*) is typed, break evaluates *form* and returns the result.

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 645).

If *tag* is omitted, it defaults to nil.

prin1 *Variable*

The value of this variable is normally nil. If it is non-nil, then the read-eval-print loop will use 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 21.2.1, page 370. 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*

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

// *Variable*

While a form is being evaluated by a read-eval-print loop, // is bound to a list of the results printed the last time through the loop.

++ *Variable*

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

+++ *Variable*

+ + + holds the previous value of + +.

****** *Variable*

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

******* *Variable*

*** holds the previous value of **.

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 30, page 624.

32.6 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 will be 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.

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.

If you are running a noninteractive crunching program, the incremental nature of garbage collection may be of no value. 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 (si: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-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.

Garbage collection is turned off if it appears to be about to run out of memory. You get a notification if this happens. You should 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 226).

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 scavenged 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:

```
Garbage collector process state: Await Flip
Dynamic (new+copy) space 3614383, Old space 0, Static 1785534,
Free space 9322496, committed 8308770, plus fudge 262144,
 times ratio (1.0) gives 8570914, leaving 751582 before flip.
Scavenging during cons: On, Idle scavenging: 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 "committed guess" plus the "fudge" is the estimate (on the high side) for the amount of free space that would be needed to complete a garbage collection. 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 will depend 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:

```
Garbage collector process state: Await Scavenge
Dynamic space 66647, Old space 2802957, Static 2600029,
Max scavenging remaining 5372456, minimum 2635630,
Free space 8060928  2736826 might be needed for copying)
Ratio scavenging work/free space = 1.00908s0.
Scavenging during cons: On, Idle scavenging: 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 will decrease steadily, but the minimum may increase.

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. 400 (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.

32.7 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 functions login-setq and login-eval help 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. If *host* requires passwords for logging in you will be asked for a password. When logging in to a TOPS-20 host, typing an asterisk before your password will enable any special capabilities you may be authorized to use. The default value of *host* depends on which Lisp Machine you use using; it is called the associated machine (see page 662). login also runs the :login initialization list (see page 627).

Unless *inhibit-init-file* is specified as non-nil, login will load 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.

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-setq and login-eval functions below 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 will ignore 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 627), and returns **t**.

login-setq {*variable value*}... *Special Form*

login-setq is like **setq** except that it puts a **setq** form on **logout-list** to set the variables to their previous values.

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** adds the result of the form *x* to the **logout-list**.

32.8 Dribble Files

dribble-start *filename* &optional *editor-p*

dribble-start opens *filename* as a "dribble file" (also known as a "wallpaper file"). It rebinds **standard-input** and **standard-output** so that all of the terminal interaction is directed to the file as well as the terminal. If *editor-p* is non-**nil**, then instead of opening *filename* on the file computer, **dribble-start** dribbles into a Zmacs buffer whose name is *filename*, creating it if it doesn't exist.

dribble-all *filename* &optional *editor-p*

dribble-all is like **dribble-start** 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**.

dribble-end

This closes the file opened by **dribble-start** and resets the I/O streams.

32.9 Status and SStatus

The **status** and **sstatus** special forms exist for compatibility with Maclisp. Programs that wish to run in both Maclisp and Zetalisp can use **status** to determine which of these they are running in. Also, (sstatus feature ...) can be used as it is in Maclisp.

status *Special Form*

(status features) returns a list of symbols indicating features of the Lisp environment. The complete list of all symbols that may appear on this list, and their meanings, is given in the Maclisp manual. The default list for the Lisp Machine is:

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

The value of this list will be kept up to date as features are added or removed from the Lisp Machine system. 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 used. The order of this list should not be depended on, and may be different from that shown above.

This features list is used by the *# +* read-time conditionalization syntax. See page 377.

(status feature *symbol*) returns t if *symbol* is on the (status features) list, otherwise **nil**.

(status nofeature *symbol*) returns t if *symbol* is not on the (status features) list, otherwise **nil**.

(status userid) returns the name of the logged-in user.

(status tabsize) returns the number of spaces per tab stop (always 8). Note that this can actually be changed on a per-window basis, however the **status** function always returns the default value of 8.

(status opsys) returns the name of the operating system, always the symbol **:lispm**.

(status site) returns the name of the local machine, e.g. "MIT-LISPM-6". Note that this is not the *site* as described above, under (status features).

(status status) returns a list of all **status** operations.

(status sstatus) returns a list of all **sstatus** operations.

sstatus *Special Form*

(sstatus feature *symbol*) adds *symbol* to the list of features.

(sstatus nofeature *symbol*) removes *symbol* from the list of features.

32.10 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 directory of partitions is in a special block on the disk called the label. When you "cold-boot" a Lisp Machine by typing Control-Meta-Control-Meta-Rubout, the machine checks the label to see which two partitions are flagged as being the current microcode and the current system load. These are kept separate so that the microcode can be easily changed without going through the time-consuming process of generating a new 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 204"; this means that version 204 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 "92.20 Daed 1.4", meaning that this band contains version 92.20 of System and version 1.4 of Daedalus. The description is created when a Lisp world is saved away by disk-save (see below).

32.10.1 Manipulating the Label

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

Print 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 description. The two partitions that are the current load partitions, used when you cold-boot, are preceeded by asterisks.

unit may be the unit number of the disk (most Lisp machines just have one unit, numbered 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 will be printed, and the user of that machine will be notified that you are looking at his label), or the string "CC" (which will print 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.

set-current-band *partition-name* &optional (*unit* 0)
Set the current saved Lisp image partition to be *partition-name*. If *partition-name* is a number, the name LOD*n* will be 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 will offer to select the an appropriate microcode partition as well. Normally you should answer Y.

set-current-microload *partition-name* &optional (*unit* 0)
Set the current microcode partition to be *partition-name*. If *partition-name* is a number, the name MCR*n* will be 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)
Returns the name of the current band (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; this will need to be fixed using another machine's debugging hardware, giving "CC" as the *unit* argument to the functions above.

si:edit-disk-label *unit* &optional *init-p*
This 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.

print-herald &optional *format-dest*
Tells you what system versions you are currently running. This includes where it came from on the disk and what version of each system is present in your Lisp environment. *format-dest* defaults to t; if it is nil the answer will be returned as a string rather than printed out.

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.

To begin with, it tells you 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 hightes 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.

32.10.2 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, and save it away into a partition. The function load-and-save-patches does it all conveniently for you.

load-and-save-patches

Load patches and saves a band, with a simple user interface. Run this function immediately after cold booting, without logging in first; it will log in as LISPM (or whatever is specified in the site files). After loading all patches without queries, it prints the disk label and asks what band to save in. Then it saves.

If you wish to do something other than loading all and only the latest patches, you must do 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, 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*

Save 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.

It first asks you for yes-or-no confirmation that you really want to reuse the named partition. Then it tries to figure out what to put into the textual description of the label. It starts with the brief version of si:system-version-info (see page 533). Then it asks you for an "additional comment" to append to this; usually you just type a **return** here, but you can also add a comment that will be returned by **si:system-version-info** (and thus printed when the system is booted) from then on. If this doesn't fit into the fixed size available for the textual description, it asks you to retype the whole thing (the version info as well as your comment) in a compressed form that will fit. The compressed version will appear in the textual description in **print-disk-label**.

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**.

Please don't save 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.

32.10.3 Garbage Collecting to Compress Bands

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)**.

When you garbage collect to reduce the size of a band, it is best to do two garbage collections. If you process any static areas (for example, if you use **si:clean-up-static-area**), you should process the same set of static areas both times. This is to keep the data compacted toward the bottom of the virtual address space, which makes it possible to run the saved band in small PAGE partitions. Here is why it works:

The saved system band occupies a nearly contiguous range of virtual memory, at the bottom of the address space. Doing one garbage collection copies all the useful data. Since the lowest addresses were already in use before, the copy has to occupy higher addresses. The low addresses become a hole. The highest used address is now much higher than before.

The second garbage collection copies everything again. Assuming that the first garbage collection actually freed some data, the copy will fit into the hole left by the first garbage collection. The high addresses used after the first collection are now free, and the highest used address is back down to its original value.

If you do only one garbage collection and then load in more data, another garbage collection later will not have a neat hole to copy into. Fragmentation may develop. As long as two garbage collections are done in a row, there should be no problem. In any case, problems will only occur with PAGE partitions smaller than 36000 blocks.

32.10.4 Installing New Software

The version numbers of the current microcode and system are announced to the INFO-LISPM mailing list. When a new system becomes available, mail is sent to the list explaining where to find the new system and what is new about it. Sometimes a microcode and a system go together, and the new system will not work with the old microcode and vice versa. When this happens extra care is required to avoid getting incompatible loads current at the same time so that the machine will not be able to boot itself.

All of the extant microcode versions can be found on the SYS: UBIN; directory. Microcode version *nnn* is in SYS: UBIN; UCADR MCR *nnn*. To copy a new microcode version into one of the microcode load partitions, first do a **(print-disk-label)** to ensure that the partition you intend to bash is not the current one; if it was, and something went wrong in the middle of loading the new microcode, it would be impossible to cold-boot, and this is hard to fix.

Then, install the microcode (on the non-current partition) by using **si:load-mcr-file**.

si:load-mcr-file *microcode-file partition*

Load the contents of the file *microcode-file* into the designated partition. *microcode-file* is either the version number of the system microcode to be loaded, or the pathname of a file containing microcode (in "MCR" format), normally SYS: UBIN; UCADR MCR. *partition* is either the number of a MCR partition, or the name of one, such as "MCR1". This takes about 30 seconds.

The system load, unlike the microcode load, is much too large to fit in a file. Therefore, the only way to install an updated system on a machine is to copy it from another machine that already has it. So the first step is to find a machine that is not in use and has the desired system. We will call this the source machine. The machine where the new system will be installed is the target machine. You can see who is logged into which machines, see which ones 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.

The function for actually copying a system load partition off of another machine is called as follows. Before doing this, double-check the partition names by printing the labels of both machines, and make sure no one is using the source machine.

si:receive-band *source-host source-band target-band* &optional *subset-start subset-size*

Copy the partition on *source-host*'s partition named *source-band* onto the local machine's partition named *target-band*. ("Band" means "partition".) 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 puts up a notification 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 up till the end of the data in the band. These are most often useful for restarting a transfer that was aborted due to network problems or a crash, based on the count of hundreds of blocks that is printed out during the transfer.

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-partition 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 using set-current-microload and set-current-band. Double-check everything with print-disk-label. Then cold-boot the machine, and the new system should come up in a half-minute or so.

If the microcode you installed is not the same version as was installed on the source machine from which you got the system load, you will need to follow the procedure given below under "installing new microcode". This can happen if someone hasn't installed the current microcode yet on that other machine.

32.10.5 Installing New Microcode

When an existing system is to be used with a new microcode, certain changes need to be made to the system, and it should then be dumped back out with the changes. The error handler has a table of errors that are detected by microcode. The hardware/microcode debugger (CC) has a microcode symbol table. These symbols are used when debugging other machines, and are also used by certain metering programs. These tables should be updated when a new microcode is installed.

The error-handler will automatically update its table (from a file on the SYS: UBIN; directory) when the machine is booted with the new microcode. The CC symbol table is updated automatically when you do a disk-save; you can also do so by hand as follows:

```
(login 'lispm)
(pkg-goto 'cadr)
(cc-load-ucode-symbols "SYS: UBIN; UCADR SYM nnn")
(pkg-goto)
```

where *nnn* is the microcode version number. This operation will take a minute or two; after it has read in most of the file the machine will stop for a long time while it sorts the symbols. It will look like it has crashed, but it hasn't, really, and will eventually come back.

After booting the system with the new microcode and following the above procedure, the updated system should be saved with **disk-save** as explained above. Note that this operation does not change the system version number. Once the new band is verified to work, the old band can be removed from the label with **si:edit-disk-label** if desired.

32.11 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*)... *Special Form*

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 is the host into which the logical host **SYS** will translate.

: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 478). 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 will also be 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" "SRC:<L.CC>")
    ("CHAOS" "SRC:<L.CHAOS>")
    ("DEMO" "SRC:<L.DEMO>")
    ...
    ("SITE" "SRC:<L.SITE>")
    ("SYS" "SRC:<L.SYS>")
    ("SYS2" "SRC:<L.SYS2>")
    ...
    ("ZMAIL" "SRC:<L.ZMAIL>")
    ("ZWEI" "SRC:<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
: This should be t if there is a chaosnet, nil if not.

:ether
: This should be t if the Lisp machine is on an ethernet, nil if not.

:chaos-file-server-hosts
: This should be a list of names of hosts that have file servers.

: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 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 and nil.

: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.

:machines-with-local-file-systems
: This should be a list of names of lisp machines that normally have Local-

File systems.

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

32.11.1 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)
```

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.

32.11.2 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*] *Special Form*

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*] *Special Form*

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**.

32.11.3 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.

32.11.4 Initializing a Band at a New Site

To initialize the Lisp machine system after moving to a new site, it is best to bring the system on a disk with a local file system. Copy a set of site files to a directory on the local file system before you leave the old site. Although the system band will not know about any of the hosts at the new site, the local file system will run properly.

Log in on host LM and edit the site files there so that they are correct for the new site. Then use si:set-sys-host to make LM the SYS host, specifying the directory on which the edited site files reside. Then (make-system 'site 'compile) will compile and load the new site files. At this point, the new site's hosts will be accessible, and the SYS host will be set according to the site files.

Alternatively, you can use si:set-sys-host to specify some other host at the new site, even though it is not known in the host table. To do this, specify the chaosnet address of the host as the *host-address* argument. Then the system will not mind that it does not know any host by the specified name. You must also specify the *operating-system-type* argument, a keyword such as :tops20, :vms or :unix. This is because the system cannot determine the type of filename syntax to use for the host from the host table, as it usually does. *site-file-directory* must also be specified.

Now you can refer to the site files on that host using pathnames such as SYS: SITE; SITE LISP. Update the files, do make-system as above, and you are done.

si:set-sys-host *host-name* &optional *operating-system-type host-address site-file-directory default-device*

Specify the host for the SYS logical host to translate into; this is the host on which system source files, site files, patch files, error tables and microcode symbols will be found.

Normally it is enough to specify just the host name. Specifying more than one argument is useful mainly when you are operating at a new site and do not have correct host tables loaded. The additional arguments allow you to specify the information that would normally be obtained by looking for the host name in the host table. This gets you far enough to load site files from that host, which will set things up properly.