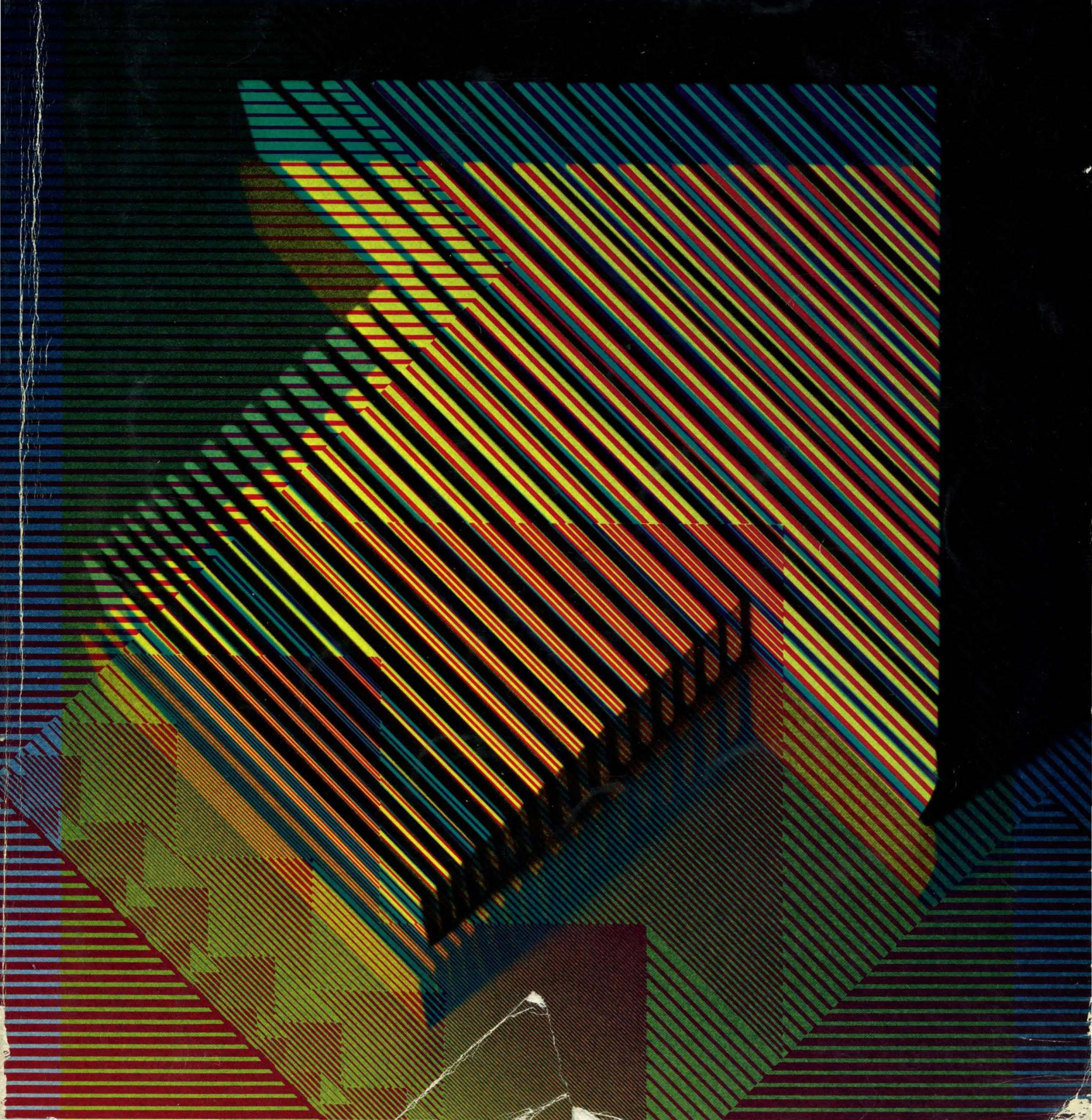
1 User's Guide to Symbolics Computers

symbolics



1 User's Guide to Symbolics Computers

symbolics

Cambridge, Massachusetts

User's Guide to Symbolics Computers

# 996015

March 1985

Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.

Text typography: Century Schoolbook and Helvetica produced on Symbolics 3600-family computers from Bitstream, Inc., outlines; text masters printed on Symbolics LGP-1 Laser Graphics Printers.
Cover design: Schafer/LaCasse
Cover printer: W.E. Andrews Co., Inc.
Text printer: ZBR Publications, Inc.

Printed in the USA.

Printing year and number: 87 86 85 9 8 7 6 5 4 3 2 1

Table of Contents

List of Figures

1. Starting up

This section provides information about how to start, cold boot, log in to, and log out of the 3600 family of machines. It assumes that the software is installed and your site has been configured. If you are not sure that this has been done, check with your site manager. The software must be installed and the site configured before you attempt to use the system. For information on installation and site configuration: See the section "Software Installation Guide" in *Installation and Site Operations*.

1.1 Getting Started

To power up and start using the your Symbolics Computer, use the following procedure:

1. If you have a 3600:

   a. Plug in the 3600. The front panel lights on the processor cabinet display "3600" when the machine is plugged in.

   b. Turn the key on the front panel to the vertical position, marked LOCAL.

   c. After the front panel lights display "Power up?", push the spring-loaded switch marked YES. The front panel lights then display "3600 on".

2. If you have a 3670 or 3640:

   a. Plug in the machine.

   b. Press the Power button on the front panel.

3. After you have turned the machine on, the FEP has control of the console. Now you Cold boot the machine.

   Cold booting is a complete reset of Lisp. It loads in a new world. Cold booting erases the contents of the Lisp environment, including the contents of editor and mail buffers. Never cold boot a machine that is being used by someone else.

   You can cold boot the machine when all of the following conditions hold:

   - The screen is white.

- The FEP prompt (`Fep>`) appears in the upper left-hand corner.

- A blinking cursor appears.

However, you cannot cold boot if any of the following conditions is true:

- The screen is black. This might indicate that the console is not turned on.

- The screen is white, but no characters appear. This might indicate that the video cable is disconnected. If the video cable is connected, this condition might indicate a malfunction of the FEP, and you should call your Symbolics Field Service Representative.

Cold boot the machine by typing the FEP command Boot at the FEP prompt (`Fep>`) and pressing RETURN.

```
on the screen:   Fep>
you type:        boot
you press:       RETURN
```

As it cold boots the machine, the FEP executes the commands in the latest version of the file named FEP0:>boot.boot. Booting takes about two minutes. When the FEP has successfully completed the cold boot, the *herald* (a multiline message beginning `Symbolics 3600 System`, or `Symbolics 3670 System` as appropriate) appears.

4. Log in.

After cold booting, you are in a window named Lisp Listener 1. You are now ready to log in. If your login name is Whit, you can log in in any of the following ways: (Note that the examples are given in upper and lower case, but the machine is not case sensitive. You can use all upper case, all lower case, or mixed case as you prefer.)

- To log into the default host machine, using your init file, type
  ```
  Login Whit
  ```

- To log into your machine, without your init file, type
  ```
  Login Whit :init file none
  ```

- To log into another machine "sc3", using your init file, type
  ```
  Login Whit :host sc3
  ```

If the host machine you log in to is a timesharing computer system, you must have a directory and account on that host machine.

- For more information about cold booting: See the section "Front-end Processor", page 197.

- For more information about logging in: See the section "Login Command", page 28.

- For more information about how to write init files: See the section "Customizing Your Environment", page 161.

1.2 Logging Out

1. Use a Lisp Listener by pressing `SELECT L`.

2. Log out by typing either the command `Logout` or the function `(logout)`.

   Wait until the Lisp Listener says that you have been logged out before you go to the next step.

3. Cold boot the machine.

   This step is optional. It is not necessary to cold boot if the machine has been used only a short while and if no major changes to the machine state have been made. If the machine has been used for several hours and many files have been loaded or read into it, we recommend that the machine be cold-booted.

   Cold booting frees up virtual memory and puts the machine in a fresh state. In this way your customizations do not affect the next user's environment.

Note: You need not turn the machine off each night; however, it does not hurt the machine to do so.

2. The Console

This chapter describes the basic characteristics of the devices that are used to talk to Symbolics Lisp Machines. These include one or more bit-raster displays, a specially extended keyboard, and a graphical input device called a *mouse*. Collectively these form a complete and extremely flexible user interface, called the *console*.

2.1 The Screen

The screen always contains one or more windows. Regardless of which windows are displayed, the screen always contains some information displays, including a *mouse documentation line* and a *status line*. These information displays are helpful in determining whether a 3600 is operating normally or needs intervention. See the section "Recovering From Errors and Stuck States", page 43.

2.2 Mouse Documentation Line

The mouse documentation line contains information about what different mouse clicks mean. As you move the mouse across different mouse-sensitive areas of the screen, the mouse documentation line changes to reflect the changing commands available.

When no documentation appears, it does not necessarily mean that the mouse clicks are undefined. Not all programs have provided material for the mouse documentation line. When the mouse documentation line is blank at "top level" in a window, the mouse usually offers some standard commands. mouse-L selects a window; mouse-R brings up either the system menu or a menu specific to the application.

The mouse documentation line is normally displayed in reverse video. Pressing FUNCTION m-C complements the video state of the mouse documentation line.

2.3 Status Line

The status line is the line of text at the bottom of the screen. It contains the following information:

- Date and time
- Login name

- Current package
- Process state
- Run bars
- Other context-dependent information, such as
    - Console idle time
    - Network service indicators

2.4 Process State

The process state refers to the processes associated with the selected window. See the section "Selecting and Creating Windows". The following list shows some common states:

State	*Meaning*
`Mouse Out`	Waiting for the mouse process to notice a change of windows.
`Net In`	Waiting for data from another machine on the network.
`Net Out`	Waiting to send data to another machine on the network.
`Open`	Waiting to open a file on another machine on the network.
`Run`	Process is running.
`Tyi`	Waiting for input from keyboard or mouse.

2.5 Run Bars

The run bars are thin horizontal lines near the process state in the status line. A description of each one follows:

GC bar (under the package name)
: Left half is visible when the scavenger is looking for references to objects that are candidates to become garbage. Right half is visible when the transporter is copying an object.

Disk bar
: Visible when the processor is waiting for the disk, typically because of paging. Nonpaging disk I/O usually waits via **process-wait**, in which case this bar does not appear.

Run bar (under the run/wait state)
: Visible when a process is running and not waiting for the disk. Not visible when the scheduler is looking for something to do.

Disk-save bar
: Visible when **disk-save** is reading from the disk; **disk-save** alternatively reads and writes large batches of pages. The alternating state of this bar tells you that **disk-save** is working while you wait for it.

2.6 The Keyboard

There are 88 physical keys on the keyboard. The keyboard has unlimited rollover, meaning that a keystroke is sensed when the key is pressed, no matter what other keys are depressed at the time.

The keys are divided into three groups: function keys, character keys, and modifier keys. Function keys and character keys transmit something. They have white labels on the tops and are typed in sequence. Modifier keys are intended to be held down while a function or character key is typed, to alter the effect of the key. They have dark labels on the tops.

Function Keys
: FUNCTION, ESCAPE, REFRESH, CLEAR INPUT, SUSPEND, RESUME, ABORT, NETWORK, HELP, LOCAL, TAB, BACKSPACE, PAGE, COMPLETE, SELECT, RUBOUT, RETURN, LINE, END and SCROLL

Character Keys
: a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9 : - = ` \ | ( ) ; ' , . / and the space bar.

Modifier Keys
: CAPS-LOCK, SYMBOL, SHIFT, REPEAT, MODE LOCK, HYPER, SUPER, META, and CONTROL

The following keys are reserved for use by the user (for example, to put custom editor commands or keyboard macros on):

CIRCLE
SQUARE
TRIANGLE
HYPER

2.7 The Mouse

The mouse is a pointing device that can be moved around on a flat surface. These motions are sensed by the Lisp Machine, which usually responds by moving a cursor around on the screen in a corresponding manner. The shape of the cursor varies, depending on context.

There are three buttons on the mouse, called left, middle, and right. They are used to specify operations to be performed. Typically you point at something with the mouse and specify an operation by clicking the mouse buttons. "Shift clicks", indicated by sh-, are conventionally distinguished from single clicks. Holding down the SHIFT key while clicking a button is the same as clicking that button twice quickly. In any specific context, there are up to six operations that can be performed with the mouse, invoked by Left, sh-Left, Middle, sh-Middle, Right, and sh-Right clicks. Some of these operations are local to particular programs such as the editor, and some are defined more widely across the system.

Typically the operations available by clicking the mouse buttons are listed at the bottom of the screen. This display, called the mouse documentation line, changes as you move the mouse around or run different programs.

Sometimes holding a mouse button down continuously for a period of time may also be defined to perform some operation, for example, drawing a curve on the screen. This will be indicated by the word "Hold". For example, "Middle Hold" means to click the middle mouse button down and hold it down, releasing it only when the operation is complete. "sh-Left Hold" means hold down the SHIFT key and click left, then release the SHIFT key but hold the left button down until the operation is complete.

3. Communicating with the Lisp Machine

3.1 Entering Commands

3.1.1 Overview of the Command Processor

The command processor is a utility program that collects arguments on behalf of a command and then runs that command for you. The command processor takes care of various chores:

- Prompting for arguments
- Checking arguments for correctness
- Providing completion when possible
- Providing documentation on request

By default, the command processor is on in all Lisp Listeners and **break** loops. The prompt "Command: " indicates that you should enter a command or a Lisp form. In the default command processor *mode*, input to a Lisp Listener or **break** loop is treated as a command if it begins with an alphabetic character. Input is treated as a Lisp form if it begins with a nonalphabetic character or is preceded by a comma.

For information on entering a command: See the section "Entering a Command", page 10.

For information on changing the command processor's mode, prompt, and other characteristics: See the section "Customizing the Command Processor", page 164.

For information on turning the command processor on and off: See the section "Turning the Command Processor on and Off", page 15.

For descriptions of predefined commands: See the section "Command Descriptions", page 16.

For information on the command processor reader and the facility for defining your own commands: See the section "The Command Processor Program Interface" in *Programming the User Interface.*

3.1.2 Parts of a Command

A command has three logical parts to it, which you specify in this order:

1. Command name. This is a word or a series of words separated by spaces.

2. Positional arguments. These are arguments that the command processor prompts for directly after the command name. Some commands have several positional arguments; others have none. Commands that have arguments might use default values for the ones that you don't specify.

3. Keyword arguments. Some commands have keyword arguments that make it simple to modify the meaning of the commands. Most of these arguments require values. These arguments have default values that the command processor assumes if you specify the command without mentioning the argument name. Some commands have arguments whose values differ according to whether you omit the argument altogether or mention the argument name and omit its value. Some keyword arguments do not have values at all.

A command actually has other logical parts as well. These are characters that serve to delimit the command, the command name, and the arguments. Still other characters serve as "commands" inside the command, providing completion and help messages.

For information on entering command names and arguments: See the section "Entering a Command", page 10. See the section "Completion in the Command Processor", page 14.

For information on help in the command processor: See the section "Help in the Command Processor", page 14.

3.1.3 Entering a Command

In entering a command, you enter the components in order: first the command name, then its positional arguments, then its keyword arguments, then the command terminator (RETURN or END). (When the command processor is in **:form-preferred** mode, you must precede the entire command by a colon: See the section "Setting the Command Processor Mode", page 165.)

The command processor can *complete* components of commands. While you are typing a command name or keyword argument name, if you press SPACE the command processor attempts to complete the current word and all previous words in that command name or keyword argument name. If you press COMPLETE, the command processor attempts to complete the entire command name or keyword argument name. The command processor can also complete argument values that are members of a limited set of possibilities. When you terminate a command, the command processor completes any command component in progress.

Some arguments have *default* values. If you press SPACE instead of typing an argument, the command processor uses the default for that argument. The command processor also uses the defaults for any arguments you haven't specified at all when you terminate the command.

All this means that you don't have to type an entire command to enter it. Suppose, for example, that you type the following:

```
d e SPACE f SPACE f o o . * SPACE : q SPACE y RETURN
```

You see the following on the screen:

```
Delete File (file [default ACME-BLUE:>joe>foo.lisp]) foo.*
(keywords) :Query (Yes, No, or Ask) Yes
```

While entering a command, pressing HELP or c-? displays documentation appropriate for the current stage of entering the command. See the section "Help in the Command Processor", page 14.

3.1.3.1 Supplying a Command Name

You type the command name, or some portion of it, followed by SPACE. The command processor either recognizes the command from what you have typed or it doesn't.

- When it recognizes the command, it fills in the part of the command name that you didn't type and then prompts you for the first argument. For example, you type:

  ```
  d e SPACE f SPACE
  ```

  The command processor displays:

  ```
  Delete File (file [default ACME-BLUE:>joe>foo.lisp])
  ```

- When it doesn't recognize what you have typed so far as being the possible beginning of a command, the command processor informs you that no such commands are available. You have to edit your input or erase it and start over.

- When it determines that what you have typed matches the beginning of several different commands, it fills in as much of the command as possible and waits for more input. You can use SPACE again to see if there is a default completion for this command, or you can use HELP or c-? to see the set of commands that begin with what you typed.

3.1.3.2 Supplying Positional Arguments to a Command

When the command processor has prompted you for a positional argument, you enter whatever argument is appropriate for the command. The prompt words indicate what the command expects:

```
Delete File (file [default ACME-BLUE:>joe>foo.lisp])
Set Package (A package)
Load Patches (for systems)
```

An argument can be either a single item or, sometimes, a set of items separated by commas. An argument cannot end with a comma, so SPACE can appear after a comma for attractiveness if you want; the command processor just ignores SPACE after a comma.

```
Load Patches (for systems) System, Zmail
```

You end each argument with SPACE. The command processor then checks whatever you have entered and prompts for the next argument (if there is one) or for the keyword arguments. If you haven't typed anything except SPACE, it fills in the default argument when one exists. Otherwise it checks what you typed for validity (for example, if the command wants a number, it makes sure that you didn't enter a string).

```
Delete File (file [default ACME-BLUE:>joe>foo.lisp]) foo.* (keywords)
```

Some arguments can only be members of a limited set of possibilities, displayed in the prompt. In this case the command processor can attempt to complete the argument. If you begin to type the argument and press SPACE, the command processor attempts to complete the current word and all words before that word in the argument. If you begin to type the argument and press COMPLETE, the command processor attempts to complete the entire argument. For example, you type:

```
s e SPACE c SPACE p SPACE f - p SPACE
```

The command processor displays:

```
Set Command Processor (Form-Only, Form-Preferred, Command-Preferred,
or Command-Only) Form-Preferred (prompt string)
```

What if one of the items in the argument list needs to contain one of the special characters (SPACE, comma, leading colon, or RETURN)? Use double quotes to delimit that item:

```
Show Hosts (hosts) Missouri,"Red River"
```

Most arguments have a default, which is usually indicated by the argument's prompt. When you want to use the default for an argument, you can indicate that simply by using SPACE. This terminates the argument, causing the command processor to fill in the default.

Sometimes when you supply a value for argument, the value that the command processor actually uses is a function of both the default and what you type. This is what happens with pathname arguments; the default pathname and the value that you type are *merged* to form the argument value that the command processor gives to the command.

Once you have specified as many of the arguments as you need (even none), you can use RETURN or END to enter the command. The command processor uses the defaults for any arguments you haven't specified.

Suppose you want to use the defaults for the remaining positional arguments, but you want to supply some keyword arguments. You must use SPACE to fill in the default for each of the remaining positional arguments. When you have finished the positional arguments, the command processor prompts for keyword arguments.

3.1.3.3 Supplying Keywords and Values for a Command

The command processor prompts for keyword arguments when you have entered all of the positional arguments for the command.

Suppose you have supplied all of the arguments to the Delete File command and are now being prompted for any keywords for modifying the standard action of the command. You enter keywords and their values in any order, finishing off the command with RETURN or END. The keyword prompt does not appear for every keyword, as that would clutter up your command.

The command processor can attempt to complete keyword argument names and values that are members of a limited set of possibilities. When you are typing a word, if you press SPACE the command processor attempts to complete that word and all previous words in the current keyword argument name or values. If you press COMPLETE, the command processor attempts to complete the entire keyword argument name or value in progress. For example, you type the following:

```
d e SPACE f SPACE f o o . * SPACE : e SPACE n SPACE
: q SPACE a RETURN
```

The command processor displays:

```
Delete File (file [default ACME-BLUE:>joe>foo.lisp]) foo.*
(keywords) :Expunge (Yes, No, or Ask) No :Query (Yes, No, or Ask) Ask
```

Most keyword arguments have several values, but some are *flag* keywords with no value. For these keywords, you do not specify a value. Often, such flag keywords exist as synonyms for some other keyword/value combination. For example, suppose there was a keyword argument called :Expunge that had three values: Yes, No, and Ask. The person defining the command could have decided for convenience to offer :No-Expunge as a flag keyword that is synonymous with ":Expunge No".

Some commands have keyword arguments with interesting defaulting behavior. These arguments have two different kinds of "defaults", one that applies when you mention the keyword without explicitly supplying a value, and one that applies when you omit the keyword altogether. For example, consider the :Expunge argument for Delete File. When you omit :Expunge, the command processor assumes you mean ":Expunge No". When you supply :Expunge and use SPACE to fill in the default, it assumes you mean ":Expunge Yes". This style of argument occurs less often than the one with the conventional defaulting behavior.

Keywords can be specified at most once in a command line. The command processor views a command line in which the same keyword has been specified twice as ambiguous; you have to correct the problem by removing one of the keyword argument pairs.

3.1.4 Editing a Command

The command processor uses the input editor to manage typing, displaying, and editing of a command that you are entering. You can move from field to field within a command, change arguments, delete keywords, even change the command name.

3.1.5 Help in the Command Processor

Press HELP to the command processor at any time before or during entering a command. (Once you have started to enter a command, you can also use c-?.) It provides documentation that is appropriate for the particular stage you have reached in entering the command.

Before starting	Explains how to enter a command processor command.
Command name	Shows the commands that could be completions of what you have typed so far.
Positional argument	Explains the characteristics of the argument that is required at this position, including possible values.
Keyword argument name	If you have not yet typed a keyword flag character, the command processor lists the remaining arguments and briefly describes them. If you have typed a keyword flag character, the command processor shows the keywords that could be completions of what you have typed so far.
Keyword argument value	The command processor presents documentation for the meaning of all the possible values of the argument.

3.1.6 Completion in the Command Processor

The command processor offers two kinds of completion: *partial* completion and *token* completion. A token is a command component, such as the command name or a keyword argument name.

- Partial completion: When you are typing a word in a command name or keyword argument name, if you press SPACE the command processor attempts to complete the current word and all previous words in the current command name or keyword argument name.

- Token completion: When you are typing a command name or keyword argument name, if you press COMPLETE the command processor attempts to complete the entire command name or keyword argument name in progress.

Completion is also available for argument values that are members of a limited set of possibilities, and for system and package names.

3.1.7 Command History

Command processor commands are maintained in the input editor input history, along with other input to the Lisp Listener or **break** loop. `c-m-Y` yanks the last element of the history. `c-m-0 c-m-Y` lists the elements of the history. A numeric argument to `c-m-Y` yanks the element of the history specified by the argument. See the section "Using the Input Editor".

3.1.8 Error Handling in the Command Processor

Part of the command processor's contract with the programs it serves is to collect syntactically valid arguments for the command you want to use. Thus if the command wants a numeric argument and you have entered a file spec, the command processor notices the problem, complains about the argument that you typed, moves the cursor there, and requests that you edit what you typed in order to make it appropriate for the command.

The command processor checks for errors of omission as well, warning you when you try to finish a command before specifying some argument that needs to be explicit.

In making its error warnings, the command processor prints out a diagnosis of the problem and asks you to correct your input. It never removes anything from what you have typed, since you are the best judge of how to remedy the problem.

3.1.9 Turning the Command Processor on and Off

The command processor is on by default in all Lisp Listeners and **break** loops. You can turn the command processor on and off, but normally you should have to do neither. If you want the command processor to treat input differently from the default, or if you want a prompt that is different from the default, you can change these characteristics by using the Set Command Processor command or setting special variables: See the section "Setting the Command Processor Mode", page 165. See the section "Setting the Command Processor Prompt", page 165.

For example, suppose you want the command processor to act as if it weren't there. You can use the Set Command Processor command to set the dispatch mode to **:form-only** and the prompt to the empty string. Alternatively, you can set **si:*cp-dispatch-mode*** to **:form-only** and **si:*cp-prompt*** to **nil** or the empty string. If you then want to return the command processor to its default behavior, you can set **si:*cp-dispatch-mode*** to **:command-preferred** and **si:*cp-prompt*** to **"Command: "**.

If for some reason you need to turn the command processor off completely, you can call **cp-off**.

cp-off *Function*

Turns off the command processor in all Lisp Listeners and **break** loops.

Once you call **cp-off**, you must call **cp-on** to turn the command processor back on.

cp-on &optional (*dispatch-mode* **si:*cp-dispatch-mode***) *prompt* *Function*

Turns on the command processor and sets its mode and prompt in all Lisp Listeners and **break** loops.

dispatch-mode is **:form-only**, **:command-only**, **:form-preferred**, or **:command-preferred**. For the meaning of these keywords: See the section "Setting the Command Processor Mode", page 165. This argument becomes the value of the variable **si:*cp-dispatch-mode***. The default mode is the current mode (the current value of **si:*cp-dispatch-mode***). The initial default mode is **:command-preferred**.

prompt is a prompt option for displaying the command processor prompt in Lisp Listeners and **break** loops. This argument becomes the value of the variable **si:*cp-prompt*** and is passed to the input editor as the value of the **:prompt** option. The value can be **nil**, a string, a function, or a symbol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor" in *Programming the User Interface*.

The default prompt depends on *dispatch-mode*. If *dispatch-mode* is **:command-preferred** or **:command-only**, the default prompt is "Command: ". If *dispatch-mode* is **:form-preferred** or **:form-only**, the default prompt is the empty string, and no prompt is displayed. If you supply a value of **nil** or the empty string, no prompt is displayed.

3.2 Command Descriptions

3.2.1 *Compile* Commands

Compile File Command

Compile File *file-spec keywords*

Compile the file designated in *file-spec*.

file-spec	The pathname of the file to compile. The default is the usual file default.	
keywords	can be:	
	:query	{yes no ask} Whether to ask for confirmation before compiling. The default is no.
	:load	{yes no ask} Whether to load the file after compiling. The default is yes.

:compiler	{Lisp use-canonical-type} The compiler to use. The default is use-canonical-type.

Compile System Command

Compile System *system keywords*

Compile the files that make up *system.*

system-spec	name of the system to compile. The default is the last system loaded.
keywords	can be:

:automatic answer	{yes no} Proceed as if any questions the system might ask during the compilation process are answered yes. The default is yes. This allows you to start a compilation that you know will take a long time and leave it to finish by itself without interruption for questions such as "*pathname* not found, continue anyway? (Y or N)".
:condition	{always new-source} Under what conditions to compile each file in the system. Always means compile each file. New-source means compile a file only if it has been changed since the last compilation. The default is new-source.
:new major version	{yes no ask} Whether to give your newly compiled version of the system the next higher version number. The default is yes. Giving the choice no will ask you to confirm that you really want to "prevent incrementing system major version number".
:query	{yes no ask} Whether to ask for confirmation before compiling each file. The default is no.
:simulate	{yes no} Print a simulation of what compiling would do. The default is no. Adding this keyword to your Compile System command string is the same as :simulate yes.
:version	{released latest newest use-default *version-number version-name*} Version of the system files to compile. The default is use-default, that is latest.

3.2.2 *Copy* Commands

Copy File Command

Copy File from *file-spec* to *destination-spec keywords*

Makes a copy of a file.

file-spec	The pathname of the file you want to copy.
destination-spec	The pathname of the location you want to put the file.
keywords	can be:

:byte size	A number. Byte size in which to do the copy operation.
:copy properties	*list of file properties.* The properties you want duplicated in the new file. The default is author and creation-date.
:create directories	{yes error query-each} What to do if the destination directory does not exist. The default is query-each.
:mode	{character binary default} The mode in which to perform the transfer. The default is default.
:query	{yes no ask} Whether to ask for confirmation before copying each file. The default is no.

Copy Microcode Command

Copy Microcode *version destination keywords*

Installs a version of microcode.

version	Microcode version number to copy. *version* is required (no default).
destination	FEP file specification. The default is created from the microcode version.
keywords	can be:

:update boot file	{FEP-file-spec none}. The default is the current default boot file name.

This command installs standard microde versions. It uses the Lisp command **si:install-microcode**. If you want to install a nonstandard microcode, you must use **si:install-microcode** directly.

Copy World Command

Copy World	*file destination keywords*

Makes a copy of a world load.

file	FEP file spec. *file* is required (no default). Note: Completion is not available for this file specification since the FEP file system does not support completion.	
destination	FEP file spec. Required when copying a world from the local host.	
keywords	can be:	
	:byte count	Number. The default is the length of the band.
	:start byte	Number. The default is 0.
	:update boot file	{FEP-file-spec none}. The default is the current default boot file name if *destination* is the local host.

3.2.3 *Create* Commands

Create Directory Command

Create Directory	*file-spec*
file-spec	The pathname of the directory to be created.

Create Link Command

Create Link	*pathname target keywords*

Creates an association between one pathname and a second pathname, called the target. The first pathname is linked to the target so that any references to the first pathname actually refer to the target.

pathname	The pathname you want to link from. The default is the standard file default.	
target	The pathname you want to link to. This pathname must exist. There is no default.	
keywords	can be:	
	:type	{Read-Only, Read-Write, Create-Through, All, or Use-Default} The kind of link to create. The default is Use-Default.

3.2.4 *Delete* Commands

Delete File Command

Delete File *file-spec keywords*

Deletes or marks for deletion the file *file-spec*.

file-spec	Pathname of the file to delete. The default is the usual file default. The version defaults to newest.	
keywords	can be:	
	:expunge	{yes no ask}. Whether to expunge the file. The default is no. Adding this keyword to your Delete File command is the same as :expunge yes.
	:query	{yes no ask} Whether to ask for confirmation before deleting the file. The default is no.

3.2.5 *Disable* Commands

Disable Services Command

Disable Services *service-type*

Turns off service(s) on the local machine.

service-type	{All-Services Append Chaos-Status Converse Lgp-Status Lispm-Finger Telnet Time} The service to turn off. The default is all-services.

3.2.6 *Edit* Commands

Edit Definition Command

Edit Definition *name*

Finds the definition of the object *name* and puts it in an editor buffer for you to edit.

name	Name of the object whose definition you want to edit.

Edit Directory Command

Edit Directory *directory-spec keywords*

Invokes the directory editor **dired** in Zmacs. See the section "Dired Mode in Zmacs" in *Text Editing and Processing.*

directory-spec	Pathname of the directory to edit. The default is the usual file default.	
keywords	can be:	
	:version	{all newest *number*} The default is all.

Edit File Command

Edit File *file-spec*

Enters the editor and reads in *file-spec*.

file-spec	The pathname of the file to edit. The default is the usual file default.

Edit Font Command

Edit Font *font*

Invokes the Font Editor with *font* loaded to be edited.

font	A font name. There is no default. Issuing the command with no arguments invokes the font editor with no font loaded.

Edit Namespace Object Command

Edit Namespace Object *class name keywords*

Create or modify an object in the namespace database.

class	{User Printer Network Host Site Namespace any} The kind of object to create or edit. The default is any.	
name	Name of the object to create or edit. The default is any.	
keywords	can be	
	:locally	{yes no} Whether to edit only a local copy of the information for the object. The default is no, meaning to edit the object in the central namespace database. Mentioning :locally in your command means yes, edit only a local copy.

3.2.7 *Enable* Commands

Enable Services Command

Enable Services *service-type*

Turns on service(s) on the local machine.

service-type	{All-Services Append Chaos-Status Converse Lgp-Status Lispm-Finger Telnet Time} The service to turn on. The default is all-services.

3.2.8 ***Expunge*** Commands

Expunge Directory Command

Expunge Directory *file-spec keywords*

Expunges files marked for deletion. The :query option is useful for directories containing subdirectories or if you have used a wildcard in the pathname.

file-spec	The pathname of the directory to be expunged. The default is the usual file default.	
keywords	can be:	
	:query	{yes no ask} Ask for confirmation before expunging the directory. The default is no.

3.2.9 ***Find*** Commands

Find Symbol Command

Find Symbol *name keywords*

Tries to find the symbol *name*.

name	The symbol to look for.	
keywords	can be:	
	:package	{all *package-name*} The package to search for the symbol. The default is the current package.
	:type	{all-types variable function flavor resource unbound} The type of the symbol. The default is all-types.

3.2.10 *Halt* Commands

Halt commands shut down some activity in such a way that you can resume it.

Halt GC Command

Halt GC

Turns garbage collection off.

Halt Machine Command

Halt Machine

Halt Machine stops execution of Lisp and gives control to the FEP. You can now enter Fep commands, for example, to warm or cold boot the machine.

3.2.11 *Hardcopy* Commands

Hardcopy File Command

Hardcopy File *file-spec keywords*

Sends a file to the local hardcopy device.

file-spec	The pathname of the file to be printed. The default is the usual file default.
keywords	can be:

:banner message	*string*. Title to appear on the cover page to identify the output. The default is the User ID.
:copies	*N*. The number of copies to print. The default is 1.
:delete	Whether to delete the file after it is printed. The default is no, not to delete. Adding the :delete keyword to your hardcopy command string is the same as :delete yes.
:file types	{Text, Suds-Plot, Press, Lgp, Xgp or use-canonical-type} The internal format of the contents of the file, to interpret for printing. The default is use-canonical-type.
:font	Font in which to print the file. The default is determined from the default-font attribute for the printer in the namespace database.
:format	{landscape portrait} Orientation on the paper for

	the output. Portrait is left to right across the short dimension of the paper. Landscape is bottom to top across the long dimension. The default is portrait.
:printer	The printer to use to output the file. The default is determined from your init file or from the default-printer attribute for the host in the namespace database.
:query	{yes no ask} Ask before printing each copy. The default is no.
:running head	{none numbered} Type of running head to print on the top of each page. The default is numbered.

3.2.12 *Help* Commands

Help Command

Help

Displays a list of all the available command processor commands.

3.2.13 *Initialize* Commands

Initialize Mail Command

Initialize Mail

Reloads Zmail without disturbing the rest of the system. The state of your mail on the machine is lost, so you only want to use this when your Zmail process is irreparably stuck.

Initialize Mouse Command

Initialize Mouse

Restarts the mouse process.

Initialize Time Command

Initialize Time *time keywords*

Resets the time. You can use this function to correct the time if it appears to be inaccurate.

time	A string representing date and time. The default is obtained from the network or from the calendar clock if there is no network available.

keywords	can be:	
	:time source	{status-line calendar-clock both} The Lisp Machine time sources to update. The default is both.

Initialize Time accepts most reasonable printed representations of date and time and converts them to the standard internal forms. The following are representative formats that are accepted by the parser:

```
"March 15, 1960" "15 March 1960" "3/15/60" "3/15/1960"
"3-15-60" "3-15" "15-March-60" "15-Mar-60" "March-15-60"
"1960-3-15" "1960-March-15" "1960-Mar-15"
"1130." "11:30" "11:30:17" "11:30 pm" "11:30 am" "1130" "113000"
"11.30" "11.30.00" "11.3" "11 pm" "12 noon"
"midnight" "m" "Friday, March 15, 1980" "6:00 gmt" "3:00 pdt"
"15 March 60" "15 March 60 seconds"
"fifteen March 60" "the fifteenth of March, 1960;"
"one minute after March 3, 1960"
"two days after March 3, 1960"
"Three minutes after 23:59:59 Dec 31, 1959"
"now" "today" "yesterday" "two days after tomorrow"
"one day before yesterday" "the day after tomorrow"
"five days ago"
```

Date strings in ISO standard format are accepted also. These strings are of the form "yyyy-mm-dd", where:

yyyy	Four digits representing the year
mm	The name of the month, an abbreviation for the month, or one or two digits representing the month
dd	One or two digits representing the day

Following are some restrictions on strings to be parsed:

- You cannot represent any year before 1900.
- A four-digit number alone is interpreted as a time of day, not a year. For example, "1954" is the same as "19:54:00" or "7:54 pm", not the year 1954.
- The parser does not recognize dates in European format. For example, "3/4/85" or "3-4-85" is always the same as "March 4, 1985", never "April 3, 1985". A string like "15/3/85" is an error. In such strings, the first integer is always parsed as the month and the second integer as the day.

3.2.14 *Inspect* Commands

Inspect Command

Inspect	*object*

Displays the components of *object*. This is similar to Show Object but it uses the Inspector, a window oriented program for showing data structures. It allows you to do something to that object, such as inspect it, modify it, or give it as the argument to a function. You exit from the Inspector by clicking the mouse on EXIT in the Inspector menu.

object	Any Lisp object. The default is "unspecified".

See the section "The Inspector" in *Program Development Utilities*.

3.2.15 *Load* Commands

Load File Command

Load File	*file-spec keywords*

Loads files into the current world.

file-spec	The pathname of the file to load. More than one pathname may be specified, separated by commas. The default is the usual file default.	
keywords	can be:	
	:package	*package-name* The package into which to load the file. The default is the file's "home" package.
	:query	{yes no ask} Whether to ask for confirmation before loading each file. The default is no.
	:silently	{yes no} Whether to print a line as each file is loaded.

Load Patches Command

Load Patches	*system keywords*

Loads patches into the current world for the indicated systems or for all systems.

system	{all *system-name1, system-name2* ... } The default is all.
keywords	can be:

:query	{yes no ask} Whether to ask for confirmation before loading each patch. The default is no.
:save	{*file-spec* None} The file in which to save the world with all patches loaded. Omitting this keyword means do not save the world. The default when this keyword is added to your command is None which means save the world and then prompt for a pathname.
:show	{yes no ask} Whether to print the patch comments as each patch is loaded. The default is yes.

Load System Command

Load System *system keywords*

Loads a system into the current world.

system	Name of the system to load. The default is the last system loaded.
keywords	can be:

:automatic answer	{yes no} Proceed as if any questions the system might ask during the loading process are answered yes. The default is yes. This allows you to start a loading process that you know will take a long time and leave it to finish by itself without interruption for questions such as "*pathname* not found, continue anyway? (Y or N)".
:compile	{always never new-source} Whether or not to compile the system before loading. The default is never. The mentioned default is new-source, meaning that mentioning :compile means compile if there have been changes made to the source file(s) since the last compilation.
:condition	{always never newly-compiled} Under what conditions to load each file in the system. Always means load each file. Newly-compiled means load a file only if it has been compiled since the last load. The default is always.
:declare status	{released latest *name*} Declares the status of this particular loaded system, so that it can be specified in the :version keyword of subsequent

Load System commands. This is the way to make a particular version of a system the "released" version, or to specify a particular configuration by name; for instance, a version of an application program that is intended for use with a Release 6 world load might be named "release-6". The default is latest.

:new major version	{yes no ask} Whether to give the your newly loaded version of the system the next higher version number. The default is ask.
:query	{yes no ask} Whether to ask for confirmation before loading each file. The default is no.
:simulate	{yes no} Print a simulation of what compiling and loading would do. The default is no. Adding this keyword to your Load System command string is the same as :simulate yes.
:version	{released latest newest use-default *version-number version-name*} Which version number to load. The default is use-default, that is latest.

The behavior of the :new major version keyword is determined by the setting of the :compile keyword. If :new major version is ask and you are compiling, you are asked "Prevent incrementing system major version number?" If you are not compiling the question is unnecessary and is not asked. If :new major version is no and you are compiling, the system is compiled and loaded using the same version number (this is rarely useful). If :new major version is yes, a new version number is created for the loaded system, whether or not you have one of the :compile options. (Incrementing the major version number without compiling might be useful if you have a set of patches you wish to escape from.)

3.2.16 *Login* and *Logout* Commands

Login Command

Login *user keywords*

Logs the *user* into the Lisp Machine.

user	Any string. Your user-id.
keywords	can be:

:host	{local *any-host-name*} A particular host computer. `Local` as an argument to :host is particularly useful if your namespace system is down and you wish to log in to your Lisp Machine without having it try to use the namespace database. The default comes from `login host` for your user object in the namespace database.
:init file	{default-init-file none} Whether to load your init file. The default is to load your default init file. To avoid loading your init file, use :init file none.

Logout Command

Logout *keywords*

Logs you out of the Lisp Machine.

keywords can be:

:save buffers	{yes no ask} Whether to write out modified editor buffers to disk. The default is yes.
:save mail	{yes no ask} Whether to write out modified mail buffers to disk. The default is yes.

3.2.17 *Rename* Commands

Rename File Command

Rename File *from-file to-file keyword*

from-file	The pathname of the file to be renamed. The default is the usual file default.
to-file	The new pathname. The default is the usual file default.
keyword	can be:

:query	{yes no ask} Whether to ask for confirmation before renaming. The default is no.

3.2.18 *Reset* Commands

Reset Network Command

Reset Network

Turns your network interface off and back on again. This is useful if your connections appear to be stuck and nothing is being transmitted or received.

3.2.19 *Save* Commands

Save File Buffers Command

Save File Buffers *keywords*

Saves your Zmacs file buffers on disk.

keywords can be:

:query	{yes no ask} The default is yes, meaning that it asks you about each buffer before writing it out.

Save Mail Buffers Command

Save Mail Buffers *keywords*

Saves on disk any mail buffers you have loaded.

keywords can be:

:expunge	{yes no ask} Whether to expunge each buffer before saving. The default is ask.
:query	{yes no ask} The default is yes, meaning that it asks you about each buffer before writing it out.

Save World Command

Save World *file-spec*

Saves the current world in *file-spec*.

file-spec	The pathname to use for the saved world. The default is the FEP file specification for the local machine, based on the version number of the current system.

3.2.20 *Select* Commands

Select Activity Command

Select Activity *activity*

Selects *activity* and makes it current.

Activity can be:

Activity	*Synonym*
"Common Lisp"	
Converse	
"Document Examiner"	
Editor	Zmacs
"File System Maintenance"	FSMaint
Inspector	
Lisp	
Zmail	Mail
Notifications	
Peek	
Terminal	
"Flavor Examiner"	Flavex

3.2.21 *Send* Commands

Report Bug Command

Report Bug *system name*

Sends a bug report. It starts up a bug mail window with the message header `To:` `BUG-`*system name*. If *system name* is omitted, `BUG-LISPM` is used.

3.2.22 *Set* Commands

Set commands set status variables.

Set Base Command

Set Base *number*

Sets the input and output bases to *number*.

The value of **base** is a number that is the radix in which integers are printed, or a symbol with a **si:princ-function** property. The initial value of **base** is 10. **base** should not be greater than 36.

number	A number in base 10. The default is 10.

Set Calendar Clock Command

Set Calendar Clock *time*

time	A date string.

Machines in the 3600 family have a calendar clock that operates independently of the other Lisp Machine timers. When you cold boot and the machine fails to get the time from the network, it asks you to type in the time. If the calendar clock has been set, it uses the calendar clock reading as the default for the time you specify. If the calendar clock has not been set, it offers to set it to the time you type in.

Set Calendar Clock allows you to set this clock if you need to. It accepts the standard formats for date and time. See the section "Set Time Command", page 33.

Set Command Processor Command

Set Command Processor *mode prompt-string*

Sets the mode for the command processor.

mode

form-only	Anything typed is taken as a Lisp form to be evaluated.
form-preferred	Anything typed is taken as Lisp forms unless it is preceded by a colon (:), in which case it is taken as a command processor command.
command-preferred	Anything typed is taken as a command processor command unless it begins with a left parenthesis or, in the case of a variable to be evaluated, a comma (,) or any nonalphabetic character.
command-only.	Anything typed is taken as a command processor command.

The default is command-preferred.

prompt-string Any string. The default for command-preferred and command-only modes is `Command:` . The default for form-preferred and form-only modes is " ". To include a space in your prompt, you must enclose the string in double quotes. For example:

```
Set Command Processor Command-Preferred "Command: "
```

Set Input Base Command

Set Input Base	*new-base*
new-base	Valid *n*. The default is 10.

The value of **base** is a number that is the radix in which integers are printed, or a symbol with a **si:princ-function** property. The initial value of **base** is 10. **base** should not be greater than 36.

Since the Input Base is closely linked to the Output Base, if you set one of them, you should set the other to the same value.

Set Output Base Command

Set Output Base	*new-base*
new-base	Valid *N*. The default is 10.

The value of **base** is a number that is the radix in which integers are printed, or a symbol with a **si:princ-function** property. The initial value of **base** is 10. **base** should not be greater than 36.

Since the Output Base is closely linked to the Input Base, if you set one of them, you should be sure to set the other to the same value.

Set Package Command

Set Package *package-name*

Makes *package-name* the current package; in other words, the variable **package** is set to the package named by *package-name.*

package	The name of a package.

Set Site Command

Set Site *site name*

Starts a dialogue to set the current site to be *site name.*

Set Time Command

Set Time *time-spec*

Sets the local time on your machine. This allows you to set the time that appears in the lower left hand corner of the status line if you need to.

time-spec	A date string.

Set Time accepts most reasonable printed representations of date and time and converts them to the standard internal forms. The following are representative formats that are accepted by the parser:

```
"March 15, 1960" "15 March 1960" "3/15/60" "3/15/1960"
"3-15-60" "3-15" "15-March-60" "15-Mar-60" "March-15-60"
"1960-3-15" "1960-March-15" "1960-Mar-15"
"1130." "11:30" "11:30:17" "11:30 pm" "11:30 am" "1130" "113000"
"11.30" "11.30.00" "11.3" "11 pm" "12 noon"
"midnight" "m" "Friday, March 15, 1980" "6:00 gmt" "3:00 pdt"
"15 March 60" "15 March 60 seconds"
"fifteen March 60" "the fifteenth of March, 1960;"
"one minute after March 3, 1960"
"two days after March 3, 1960"
"Three minutes after 23:59:59 Dec 31, 1959"
"now" "today" "yesterday" "two days after tomorrow"
"one day before yesterday" "the day after tomorrow"
"five days ago"
```

Date strings in ISO standard format are accepted also. These strings are of the form `"yyyy-mm-dd"`, where:

`yyyy`	Four digits representing the year
`mm`	The name of the month, an abbreviation for the month, or one or two digits representing the month
`dd`	One or two digits representing the day

Following are some restrictions on strings to be parsed:

- You cannot represent any year before 1900.
- A four-digit number alone is interpreted as a time of day, not a year. For example, `"1954"` is the same as `"19:54:00"` or `"7:54 pm"`, not the year 1954.
- The parser does not recognize dates in European format. For example, `"3/4/85"` or `"3-4-85"` is always the same as `"March 4, 1985"`, never `"April 3, 1985"`. A string like `"15/3/85"` is an error. In such strings, the first integer is always parsed as the month and the second integer as the day.

Set User Id Command

Set User ID *user-id*

Sets the user ID appearing in the status line at the bottom of the screen. This allows you to change the apparent user of the machine without logging out and logging in again, thus preserving the current environment. The new user ID appears as the author of files and in the result of a Show Users command. Zmail buffers and messages sent are not affected, so you might want to manually insert a `From:` field in any mail you send.

user-id	A string.

3.2.23 *Show* Commands

Show commands allow you to request informational displays of all kinds. These are displays that you do not interact with.

Show Command Processor Status Command

Show Command Processor Status

Displays the current mode of the Command Processor and the current prompt.

Show Directory Command

Show Directory *pathname keywords*

Displays a directory listing. The default for name, type, and version of *pathname* is **:wild**. The format of the listing varies with the operating system.

pathname	The default is the usual file default.	
keywords	can be:	
	:size	same or larger than *N* blocks. The default is 1.
	:since	a date.
	:before	a date.
	:order	{oldest-first smallest-first largest-first newest-first standard} The default is standard.

Show Disabled Services Command

Show Disabled Services

Shows you which services are disabled (with the Disable Services Command).

Show Documentation Command

Show Documentation *topic keywords*

Displays the documentation for *topic*. If you omit *topic*, you are prompted for it. If *topic* is more than one word, it must be enclosed in double-quotes:

```
Show Documentation (for topic) "The Document Examiner"
```

keywords	can be:	
	:destination	{screen or lgp}. Where to display (print) the documentation. The default is screen.

See the section "Using the Online Documentation System", page 119.

Show FEP Directory Command

Show FEP Directory *host unit*

Displays a description of the FEP files on *unit*.

host	A host on the network. The default is local.
unit	Disk structure. The default is FEP0:. *unit* can be one of the following:

- An integer smaller than 20., interpreted as a disk unit number on the local host.
- An integer larger than 19., interpreted as the Chaosnet address of a remote host. Displays the contents of unit 0 on that host.
- A symbol, interpreted as the name of a remote host. Displays the contents of unit 0 on that host.
- A string of the form "*host|unit*", where *host* is the name or Chaosnet address of a remote host and *unit* is an integer representing a disk unit number on that host.

Show FEP Directory first displays an estimate of the number of free blocks and the proportion of blocks used on *unit*. It then displays a summary of the files on *unit*, one line per file. For each file, it displays the file name, the length in blocks and in bytes, the byte size, the creation date, the comment, and the author.

Show File Command

Show File *file-spec*

Displays a file on the screen. If there is more than one screenful, it pauses between screenfuls displaying `--More--` at the bottom.

file-spec	The pathname of the file to be printed. The default is the usual file default.

Show Font Command

Show Font *font*

Displays all characters of the font. You can get a list of the fonts loaded by clicking on List Fonts in the font editor. You enter the font editor by using the edit font command with no arguments.

font	Font name.

Show GC Status Command

Show GC Status

Displays statistics about the garbage collector.

Show Herald Command

Show Herald *keywords*

Displays the herald message. The herald is part of the screen display on a cold booted machine. It shows you the name of the FEP file or partition for the current world load, any comment added to the herald, a measure of the physical memory and swapping space available, the versions of the systems that are running, the site name, and the machine's own host name.

keywords	can be:	
	:detailed	{yes no} Whether or not to print the version information in full detail. The default is no.

Show Hosts Command

Show Hosts *host-spec keywords*

Asks each of the *hosts* for its status, and prints the results. If no hosts are specified, all hosts on the Chaosnet are asked. Hosts can be specified by either name or octal number.

For each host, a line is displayed that either says that the host is not responding or gives metering information for the host's network attachments. If a host is not responding, probably it is down or there is no such host at that address. A Lisp Machine can fail to respond if it is looping inside **without-interrupts** or paging extremely heavily, such that it is simply unable to respond within a reasonable amount of time.

host-spec	List of hosts (names or numbers) or sites. The default is the local Chaosnet.

Show Legal Notice Command

Show Legal Notice

Displays the Symbolics Legal Notices, such as copyrights and trademarks.

Show Mail Command

Show Mail *file-spec*

Displays your mail inbox on the screen. If there is more than one screenful, it pauses between screenfuls displaying `--More--` at the bottom.

file-spec	The pathname of the mail inbox to be read. The default is the default inbox.

Show Notifications Command

Show Notifications *keywords*

Re-displays any notifications that have been received. Notifications are asynchronous messages from the Lisp Machine system.

keywords can be:

:before	A date to serve as one limit for notifications to show: `:before 11/1/84`
:through	A number to use as the last notification to show: `:through 17`
:matching	A string to search for. Only show notifications that contain that string: `:matching hardcopy`
:newest	A number of notifications to show, for instance, the ten most recent ones: `:newest 10`
:oldest	A number of notifications to show, for instance, the ten earliest ones: `:oldest 10`
:from	A number to use as the first notification to show.
:since	A date to serve as one limit for notifications to show.

Show Object Command

Show Object *name keywords*

Show Object tries to tell you all the interesting information about any object (except for array contents). Show Object knows about areas, structures, packages, pathnames, systems, variables, functions, flavors, and resources. It displays the attributes of each. Show Object *symbol* will tell you about *symbol*'s value, its definition, and each of its properties.

name	Any Lisp object.
keywords	can be:

:type	{all area structure partition package logical-host pathname system variable function flavor resource}. The default is all.

Show System Modifications Command

Show System Modifications
system-name keywords

With no arguments, Show System Modifications lists all the systems present in this world and, for each system, all the modifications that have been loaded into this world. For each modification it shows the major version number (which will always be the same since a world can only contain one major version), the minor version number, and an explanation of what the modification does, as entered by the person who made it.

If Show System Modifications is called with an argument, only the modifications to *system-name* are listed.

system-name	The system for which to show modifications. The default is All.
keywords	can be:

:author — A name. Show modifications by a particular person. For example:

```
:show modifications system :author kjones
```

would only show those modifications made by the person whose user ID is kjones.

:before — A date to serve as one limit for modifications to show:

```
:before 11/1/84
```

:through — A number to use as the last modification to show:

```
:through 17
```

:matching — A string to search for in the comments and only show modifications whose comment contain that string:

```
:matching namespace
```

:newest	A number of modifications to show, for instance the ten most recent ones: `:newest 10`
:number	A number. Show only this particular modification. For example: `Show Modifications :number 6` would show modification number 6.
:oldest	A number of modifications to show, for instance the ten earliest ones: `:oldest 10`
:from	A number to use as the first modification to show.
:since	A date to serve as one limit for modifications to show.

Show Users Command

Show Users *host keywords*

Shows the users logged into *host.*

host-spec	Host name or all. The default is all.	
keywords	can be:	
	:format	{brief standard detailed} How much information to display. The default is brief. Adding :format to your command means standard.
	:destination	{typeout pop-up-window} Where to display the information. The default is typeout.

3.2.24 *Start* Commands

Start GC Command

Start GC *keywords*

Turns on the garbage collector.

keywords	can be:	
	:dynamic	Dynamic Level of incremental GC.

:ephemeral	Ephemeral Level of incremental GC.
:immediately	Perform a complete garbage collection right now.

3.2.25 *Undelete* Commands

Undelete File Command

Undelete File *file-spec keywords*

Undeletes a deleted file, if the host on which the file resides supports undelete. It prompts for the name of a file to undelete. It displays a message if the specified file does not exist.

file-spec	The pathname of the file to be undeleted. The default is the usual file default.
keywords	can be:

:query	{yes no ask} The default is no.

4. Recovering From Errors and Stuck States

4.1 Introduction

Sometimes it is hard to know whether or not your machine is in trouble, because some operations, particularly those involving other network machines, can take a long time. Periodically check the process state and the run bars on the status line. The run bars flicker when the machine is working. As long as the run bars are flickering and the process state is changing occasionally, the machine is probably working properly. Some process states mean trouble if they persist, say, for a minute or more.

Look at the clock in the status line. If the clock is ticking, processes are being scheduled. If the clock is not ticking, the 3600 is halted. As long as the FEP is working, it prints a message near the top of the screen when the 3600 has halted and then gives its `Fep>` prompt. When the 3600 resumes its previous state, it updates the clock with the correct time.

4.2 Recovery Procedures

If the status line displays one of the following process states, recover by using the appropriate procedure:

State	*Recovery procedure*
`Wait Forever`	Select a different window, then reselect the one you were in.
`Output Hold`	Press FUNCTION ESCAPE (the ESCAPE key is in the top row, second from the left); if that puts you in the Debugger, use ABORT.
`Arrest`	Press FUNCTION - A (that is, a three-key sequence).
`Lock`	Try FUNCTION 0 S to see if any windows want to type out. If that does not help, press c-ABORT.
`Selected`	Press FUNCTION 0 S.
`(no window)`	Use the mouse or SELECT key to select the window you want.

You can press SUSPEND to get to a Lisp read-eval-print loop. You can press c-m-SUSPEND to force the current process into the Debugger.

4.3 The Debugger: Recovering From Errors and Stuck States

Errors that are not caught and handled by the program that triggered them invoke the Debugger. See the section "Entering the Debugger", page 192.

4.4 Resetting the FEP

Resetting the FEP restarts the FEP system, thereby discarding knowledge of the FEP's free storage area. Resetting might be necessary if you unplug the console video cable from either end or turn the console off and on. You also need to reset the FEP if you receive the error message: `Request for` *N* `longs failed`. You can reset the FEP from either the keyboard or the processor front panel.

- To reset the FEP from the keyboard:
  1. Type the form `(si:halt)` to stop the computer and give control of the keyboard to the FEP.
  2. Type the command Reset Fep to the FEP prompt.

  Alternatively, if no Lisp Listener is responsive:
  1. Press h-c-*upper-left* to stop the computer and give control of the keyboard to the FEP.
  2. Type the command Reset Fep.

  Press y to answer the confirmation prompt.

- To reset the FEP from the processor front panel:
  1. Push the red RESET button on the processor front panel.
  2. Press the spring-loaded YES switch to answer the "Reset FEP?" question.

After you reset the FEP, the keyboard is connected to the FEP, not to Lisp. Give the Start command and press RETURN to warm boot the machine and Lisp, and return control of the keyboard to Lisp.

4.5 Warm Booting

If an error occurs in the keyboard process, window system, or scheduler, making the machine unresponsive to the keyboard, you may have to warm boot the machine. Warm booting causes either a **:flush** or **:reset** message to be sent to all processes in the system, depending on the type of process.

To warm boot the computer, use the following procedure:

1. Type one of the following to a Lisp Listener:

   - Halt Machine

   - `(si:halt)`

   You are now connected to the FEP.

   If you cannot obtain a Lisp Listener window or if no Lisp Listener is responding to keyboard input, you should use h-c-*upper-left*, (*upper-left* is the key in the upper left corner of the keyboard. It corresponds to LOCAL on old keyboards and FUNCTION on new keyboards.)

2. Type `start` at the FEP prompt (Fep>) and press RETURN.

Sometimes, the prints prints `lisp stopped itself` and returns control to the FEP. When this happens, at the FEP prompt (Fep>) you should type `show status`, check the information it provides, and then type `start`.
For more information about **:flush** and **:reset**: See the section "Bashing the Process" in *Internals, Processes, and Storage Management*.

4.6 Halting

Halting the 3600 leaves all Lisp states intact. To halt the 3600 in order to connect to the FEP, type **si:halt** to a Lisp Listener or use h-c-*upper-left*. You are now connected to the FEP. To return to Lisp, type `continue` at the FEP prompt (Fep>) and press RETURN.

```
on the screen:   Fep>
you type:        continue
you press:       RETURN
```

The 3600 can halt itself under exceptional conditions. In this case, try typing `continue`. If `continue` does not work, use `start`.

5. Creating and Manipulating Files

5.1 Overview

Zmacs, the Lisp Machine editor, is built on a large and powerful system of text-manipulation functions and data structures, called *Zwei*.

Zwei is not an editor itself, but rather a system on which other text editors are implemented. For example, in addition to Zmacs, the Zmail mail reading system also uses Zwei functions to allow editing of a mail message as it is being composed or after it has been received. The subsystems that are established upon Zwei are:

- Zmacs, the editor that manipulates text in files
- Dired, the editor that manipulates directories represented as text in files
- Zmail, the editor that manipulates text in mailboxes
- Converse, the editor that manipulates text in messages

Since these subsystems share Zwei in the dynamically linked Lisp environment, many of the commands available as Zmacs commands are available in other editing contexts as well.

In this manual, we discuss Zmacs commands in the context of Zmacs only. We also describe Dired, the directory editor, since it is used within Zmacs.

5.2 Introduction to Entering Zmacs

You can enter, or invoke, the editor in several ways: Press SELECT E, use the mouse, or run either the function **ed** or the function **zwei:edit-functions**. You can also use the command Select Activity, specifying either Zmacs or Editor as its argument.

5.2.1 Entering Zmacs with SELECT E

You can invoke the editor by pressing the SELECT key and then the letter E:

- If you have already been in the editor since booting the machine, Zmacs returns you to the same place in the same buffer that you last used.
- If this is the first time you are entering Zmacs since booting the machine, Zmacs puts you in an empty buffer named `*Buffer-1*`.

SELECT E enters or returns you to the editor from anyplace in the system, not just when you are talking to Lisp.

5.2.2 Entering Zmacs with the Mouse

You can invoke the editor using the mouse.

Summon a System menu by clicking right twice [(R2)]. Then click left on the Edit option [Edit (L)], which puts you into a Zmacs buffer. As for SELECT E, if you are returning to the editor Zmacs puts you back at the same place in the same buffer, and if you are entering Zmacs for the first time it puts you in an empty buffer.

5.2.3 Entering Zmacs with ed

The Lisp function **ed** enters Zmacs from a Lisp Listener. See the function **ed**, page 158.

When reentering Zmacs within a login session, **ed** enters the editor, preserving its state as it was when you left. When entering Zmacs for the first time during a login session, **ed** initializes Zmacs and creates an empty buffer.

arg can have these values.

Value	*Description*
t	The **ed** function enters the editor, creates an empty buffer, and selects it.
Pathname or string	The **ed** function enters the editor and finds or creates a buffer with the specified file in it.
Defined symbol	The editor tries to find the source definition of that symbol for you to edit. A defined symbol can be, for example, a function, macro, variable, flavor, or system.
The symbol **zwei:reload**	The system reinitializes the editor. This destroys all existing buffers, so use this only if you have to.

5.2.4 Entering Zmacs with zwei:edit-functions

The Lisp function **zwei:edit-functions** also enters Zmacs from a Lisp Listener.

zwei:edit-functions Function

zwei:edit-functions is like **ed** in that inside the editor process it throws you back into the editor, whereas from another process it just sends a message to the editor and selects the editor's window. **zwei:edit-functions** gives *spec-list* to the editor in the same way that Edit Callers and similar editor commands would. See the section "The Zmacs Edit Callers Commands" in *Text Editing and Processing.*

This command is useful when you have collected the names of things that you need to change, for example, using some program to generate the list. *spec-list* is a list of definitions; these are either function specs (if the definitions are functions) or symbols.

Zmacs sorts the list into an appropriate order, putting definitions from the same file together, and creates a support buffer called `*Function-Specs-to-Edit-`*n*`*`. It selects the editor buffer containing the first definition in the list.

5.3 Commands

Zmacs *commands* are Lisp functions that perform the editing work. Every Zmacs command has a *name*, and many commands are bound to keys. When a command is bound to a *keystroke combination*, you invoke it by pressing those keys. For example, the Forward Word command is invoked by typing the keystroke m-F. When a command is not bound to a set of keystrokes, Zmacs calls it an *extended* command and you invoke it using its name preceded by m-X. For example, the command View Mail, an extended command, is invoked by View Mail m-X.

Command tables assign keystrokes and names to commands. Each time you press a key, Zmacs looks up the function associated with that key. For ordinary characters, the function **com-standard**, in the standard command table, inserts the character once.

5.4 Extended Commands

Extended commands extend the range of commands past the one-or-two-keystroke limitation. You invoke Zmacs extended commands by name using the m-X command:

m-X Extended Command

Prompts for the name of a Zmacs command and executes that command.

Command completion is provided.

See the section "Completion in the Minibuffer".

5.5 Keystrokes

A keystroke has a character component and a modifier component, and is performed by pressing a *primary key* (alphanumeric), possibly while holding down a *shift key* or a group of shift keys. The primary key held down with either the SHIFT or SYMBOL keys determines the *character* part of a keystroke. Whether you hold down the other shift keys, CONTROL, META, HYPER, and SUPER, determines the *modifier* part of the keystroke.

In general, commands that begin with a CONTROL (c-) key modifier operate on single characters, commands that begin with a META (m-) key modifier operate on words, sentences, paragraphs, and regions, and commands that begin with a CONTROL META (c-m-) modifier operate on Lisp code.

Prefix character commands consist of more than one keystroke per command. For example, to invoke the command c-X F, you first type the prefix character c-X and then the primary key F. Prefix character commands are not case-sensitive — that is, Zmacs converts a lowercase character following a prefix character command (like c-X) to uppercase. For example, c-X f is equivalent to c-X F.

Zmacs commands are self-delimiting. Unless otherwise specified, you do not need to type a carriage return or other terminating character to finish typing a command.

5.6 Introduction to Moving the Cursor

5.6.1 Description of Moving the Cursor

To do more than insert characters, you have to know how to move the cursor.

For complete descriptions of the commands summarized here and other cursor-moving commands: See the section "Moving the Cursor in Zmacs" in *Text Editing and Processing*.

5.6.2 Summary of Moving the Cursor

c-A	Beginning of Line
Moves to the beginning of the line.	
c-E	End of Line
Moves to the end of the line.	
c-F	Forward
Moves forward one character.	
c-B	Backward
Moves backward one character.	
m-F	Forward Word
Moves forward one word.	
m-B	Backward Word
Moves backward one word.	
m-E	Forward Sentence
Moves to the end of the sentence in text mode.	
m-A	Backward Sentence
Moves to the beginning of the sentence in text mode.	
c-N	Down Real Line
Moves down one line.	

c-P Moves up one line.	Up Real Line
m-] Moves to the start of the next paragraph.	Forward Paragraph
m-[Moves to the start of the current (or last) paragraph.	Backward Paragraph
c-X] Moves to the next page.	Next Page
c-X [Moves to the previous page.	Previous Page
c-V, SCROLL Moves down to display the next screenful of text.	Next Screen
m-V, m-SCROLL Moves up to display the previous screenful of text.	Previous Screen
m-< Moves to the beginning of the buffer.	Goto Beginning
m-> Moves to the end of the buffer.	Goto End

5.7 Introduction to the Motion Commands

Zmacs word, sentence, and paragraph motion commands all have strict definitions for where words, sentences, and paragraphs begin and end. You can modify all these definitions.

5.8 Introduction to Zmacs Help

Zmacs has many features that provide information about Zmacs commands, existing code, buffers, and files. Two features are generally useful: the HELP key and completion. See the section "Getting Help in Zmacs" in *Text Editing and Processing*.
Pressing the HELP key in a Zmacs editing window gives information about Zmacs commands and variables. For descriptions of Zmacs variables: See the section "How to Specify Zmacs Variable Settings" in *Text Editing and Processing*. The kind of information it displays depends on the key you press after HELP.

HELP ?	Displays a summary of HELP options.
HELP A	Displays names, key bindings, and brief descriptions of commands

	whose names contain a string you specify. (The A refers to *apropos*, the name of the function that finds the commands and displays their documentation.)
HELP C	Displays the name and description of a command bound to a key you specify.
HELP D	Displays documentation for a command whose name you specify.
HELP L	Displays a listing of the last 60 keys you pressed.
HELP U	Offers to undo the last major Zmacs operation, such as sorting or filling, when possible.
HELP V	Displays the names and values of Zmacs variables whose names contain a string you specify. For descriptions of Zmacs variables: See the section "How to Specify Zmacs Variable Settings" in *Text Editing and Processing*.
HELP W	Displays the key binding for a command you specify. (The W refers to where.)
HELP SPACE	Repeats the last HELP command.

Some Zmacs operations require you to provide names — for example, names of extended commands, Lisp objects, buffers, or files. Often you do not have to type all the characters of a name; Zmacs offers *completion* over some names. When completion is available, the word `Completion` appears in parentheses above the right side of the minibuffer.

You can request completion when you have typed enough characters to specify a unique word or name. For extended commands and most other names, completion works on initial substrings of each word. For example, m-X c SPACE b is sufficient to specify the extended command Compile Buffer. SPACE, COMPLETE, RETURN, and END complete names in different ways. Press HELP or click right once, [(R)], on the editor window or minibuffer to list possible completions for the characters you have typed. c-/ displays every command that contains the substring.

SPACE	Completes words up to the current word.
HELP or c-?	Displays possible completions in the typeout area.
[(R)]	Pops up a menu of possible completions.
c-/	Runs Apropos for each of the partially typed words in the name.
COMPLETE	Completes as much as possible. This could be the full name.
RETURN or END	Confirms the name if possible, whether or not you have seen the full name.

5.9 Introduction to Inserting Text

To insert new text anywhere in the buffer, position the cursor at the place you want the new text to go and type the new text. Zmacs always inserts characters at the cursor. The text to the right of the cursor is pushed along ahead of the text being inserted.

5.10 Introduction to Erasing Text

5.10.1 Description of Erasing Text

Most commands that erase text from the buffer save it so that you can get it back if you change your mind, or move or copy it to other parts of the buffer. These commands are known as *kill* commands. The rest of the commands that erase text do not save it; they are known as *delete* commands. The delete commands include c-D and RUBOUT, which delete only one character at a time, and those commands that delete only spaces or line separators. (However, when given a numeric argument, c-D and RUBOUT do save that sequence of deleted characters on the kill ring.) Commands that can destroy significant amounts of information generally kill. The commands' names and individual descriptions use the words "kill" and "delete" to say which they do.

If you issue a kill command by mistake, you can retrieve the text with c-Y, the Yank command. For details on killing and retrieving text: See the section "Working with Regions in Zmacs" in *Text Editing and Processing*.

5.10.2 Summary of Erasing Text

c-D — Delete Forward
Deletes the character after point.

RUBOUT — Rubout
Deletes the character before point.

m-D — Kill Word
Kills forward one word.

m-RUBOUT — Backward Kill Word
Kills backward one word.

m-K — Kill Sentence
Kills forward one sentence.

c-X RUBOUT — Backward Kill Sentence
Kills backward one sentence.

c-K — Kill Line
Kills to the end of the line or kills an end of line.

c-W Kill Region
Kills region (from point to mark).

c-m-K Kill Sexp
Kills forward over exactly one Lisp expression.

c-m-RUBOUT Backward Kill Sexp
Kills backward over exactly one Lisp expression.

m-\ Delete Horizontal Space
Deletes any spaces or tabs around point.

c-X c-O Delete Blank Lines
Deletes any blank lines following the end of the current line.

m-^ Delete Indentation
Deletes RETURN and any indentation at front of line.

5.11 Creating a Buffer

Zmacs creates your initial buffer when you first enter the editor. To create other buffers, use c-X B, Select Buffer, to create an empty buffer or c-X c-F, Find File, to create either an empty buffer or a buffer containing a file.

c-X B prompts for the name of the buffer to which you want to go. Type the buffer name and RETURN. If the buffer exists, Zmacs switches to that buffer and displays it on the screen. If the buffer does not already exist, Zmacs offers to let you create it by terminating the buffer name with c-RETURN. When you create a new (empty) buffer, the display is blank.

The other way to create another buffer is c-X c-F, Find File. (c-X c-F) is described in detail in "Editing Existing Files".) c-X c-F prompts for the name of a file, terminated by RETURN.

When you type c-X c-F for the first time in a Zmacs session, Zmacs offers you, as a default file name, an empty file (with the Lisp suffix native to your host computer) in your home directory on your host computer. For example:

System	*Empty Buffer Name*
Lisp Machine	foo.lisp
UNIX	foo.l
VMS	foo.lsp

Base and Syntax Default Settings for Lisp

When you read a file that has a Lisp file type into the buffer, if that file does not begin with an attribute line containing Base and Syntax attributes, Zmacs warns that the file "has neither a Base nor a Syntax attribute" and announces that it will use the defaults, Base 10 and Zetalisp. See the section "Buffer and File Attributes".

Buffer Contents with c-X c-F

The first time you use c-X c-F, you can create an empty buffer using the Zmacs default file name, create an empty buffer using a name that you specify, or create a buffer containing an existing file:

- To create an empty buffer with the initial default file name as the one Zmacs associates with your buffer, press RETURN.
- To create a new empty buffer, respond with any name. Zmacs switches to an empty buffer, gives the buffer the new name, and displays (New File) in the echo area.
- To create a new buffer containing some file, respond to the prompt with the name of that file. Zmacs switches to an empty buffer, reads that file in, and names the buffer appropriately.

5.12 Creating a File

The first time you save or write the buffer, Zmacs creates the new file. You can create a new file with c-X c-S. Since a new file does not have a name associated with it yet, Zmacs asks for a name for the new file. It offers a *default pathname*, which is the name of the buffer. If you wish to save the file out to the default pathname, simply type a RETURN in response to the prompt.

If you wish to save the buffer in another file, provide that name as your response. Completion is offered to simplify your response.

You can also write the buffer out with c-X c-W, Write File. Zmacs prompts in the minibuffer for the name of the place you want to write the buffer's contents. c-X c-W also offers a default pathname, in this case, the name you supplied with c-X c-F.

6. Sending and Receiving Messages and Mail

6.1 Using Zmail

6.1.1 Introduction

Zmail is a display-oriented mail system for the Lisp Machine. Using Zmail, you can send and receive mail, archive your mail in disk files, and operate on groups of messages selected according to very flexible criteria.

Since messages are typed into editor buffers, some familiarity with the editor is also helpful. (See the section "Zmacs Manual" in *Text Editing and Processing.*)

6.1.2 Starting up Zmail

Before running Zmail, be sure that you are logged in. See the section "Getting Started", page 1.

To run Zmail, do one of the following:

- Give the command Select Activity Zmail (or Select Activity Mail).
- Press SELECT M.

You get a display similar to Figure 1, called the *top-level display*. At this time you can send or read mail.

The top-level display, with a mail file read in, is shown in Figure 2. It consists of four windows: the Summary window, the Command menu, the Message window, and the Minibuffer, which contains the Mode line.

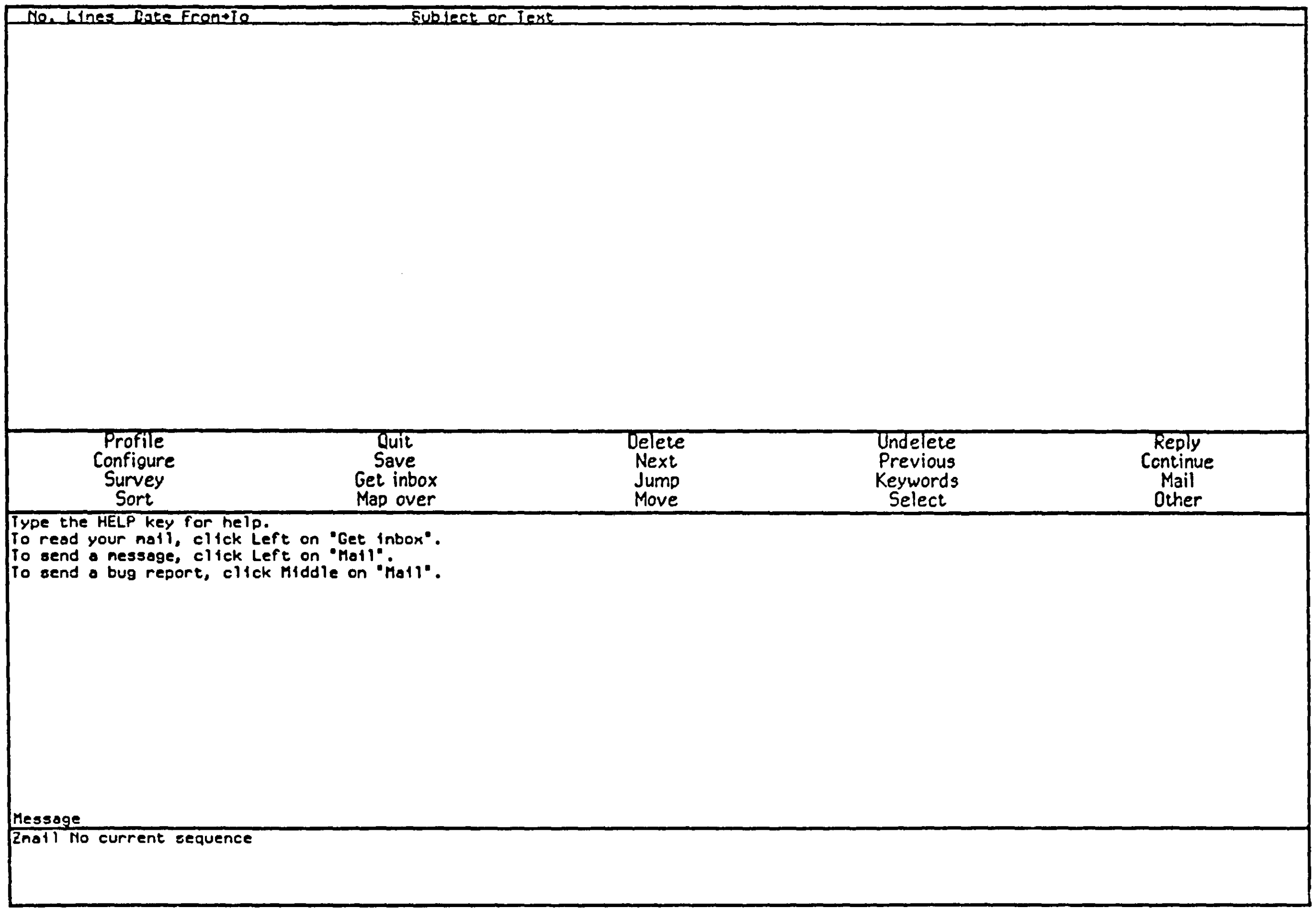
6.1.2.1 Summary Window

The *Summary Window* displays a line for each message in the current sequence, with an arrow indicating the current message.

The information provided in the summary line is:

No.	The *message number*. Whenever Zmail displays a list of messages, it numbers them for easy reference. The numbers refer only to the position of the message in the list, so when you list subsets of the mail file, the messages show up with different numbers. And when you delete or rearrange messages, the numbers change accordingly.
status letter	The *status letter* is the letter or symbol following the message number. Possible status letters are:

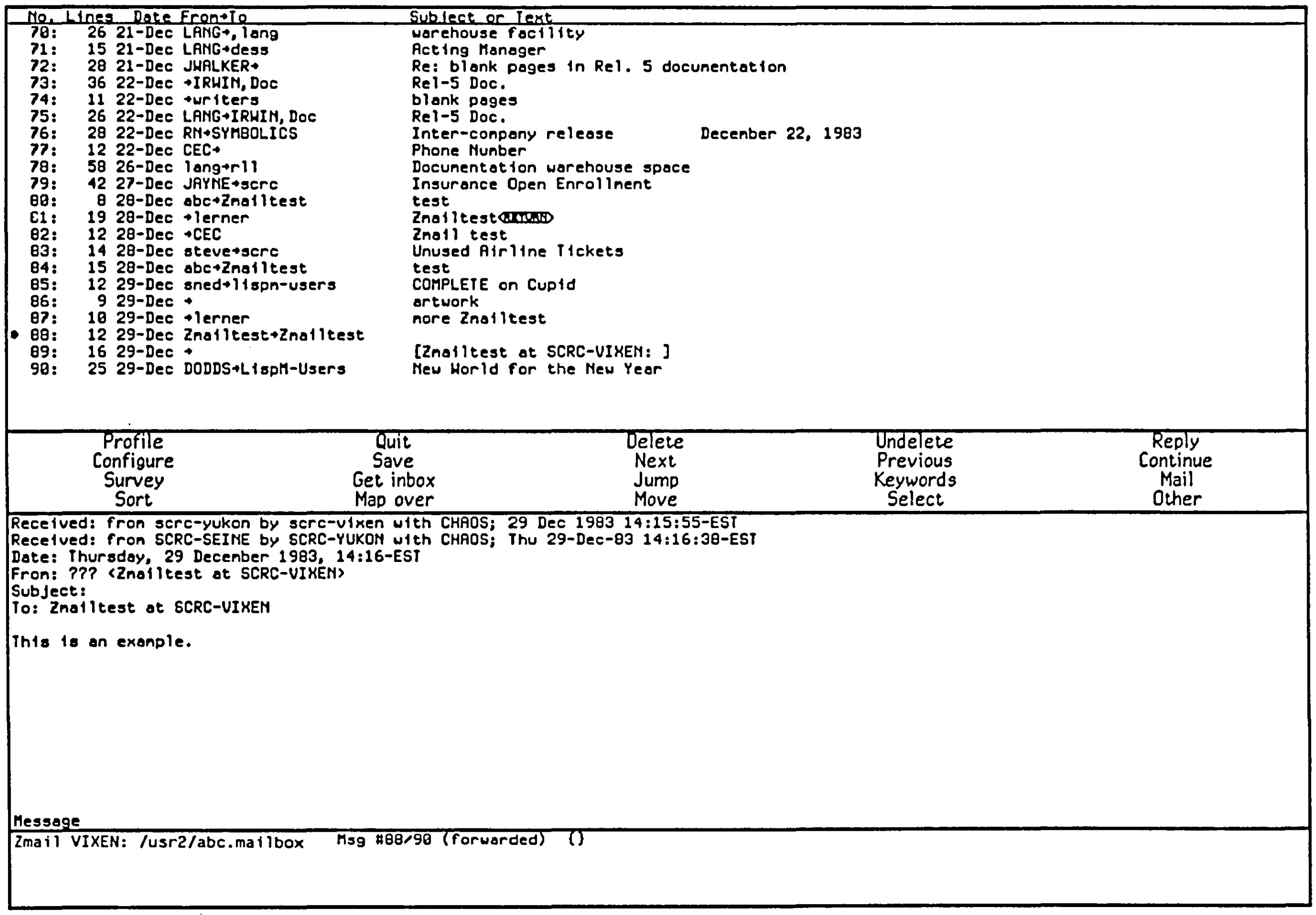
The message has not yet been seen.



```
Type the HELP key for help.
To read your mail, click Left on "Get inbox".
To send a message, click Left on "Mail".
To send a bug report, click Middle on "Mail".
```

```
Zmail No current sequence
```

Figure 1. Top-level Display



```
 No. Lines  Date From→To                   Subject or Text
  70:   26 21-Dec LANG→,lang               warehouse facility
  71:   15 21-Dec LANG→dess                Acting Manager
  72:   28 21-Dec JWALKER→                 Re: blank pages in Rel. 5 documentation
  73:   36 22-Dec →IRWIN,Doc               Rel-5 Doc.
  74:   11 22-Dec →writers                 blank pages
  75:   26 22-Dec LANG→IRWIN,Doc           Rel-5 Doc.
  76:   28 22-Dec RM→SYMBOLICS             Inter-company release       December 22, 1983
  77:   12 22-Dec CEC→                     Phone Number
  78:   58 26-Dec lang→rll                 Documentation warehouse space
  79:   42 27-Dec JAYNE→scrc               Insurance Open Enrollment
  80:    8 28-Dec abc→Zmailtest            test
  81:   19 28-Dec →lerner                  Zmailtest(RETURN)
  82:   12 28-Dec →CEC                     Zmail test
  83:   14 28-Dec steve→scrc               Unused Airline Tickets
  84:   15 28-Dec abc→Zmailtest            test
  85:   12 29-Dec smed→lispm-users         COMPLETE on Cupid
  86:    9 29-Dec →                        artwork
  87:   10 29-Dec →lerner                  more Zmailtest
● 88:   12 29-Dec Zmailtest→Zmailtest
  89:   16 29-Dec →                        [Zmailtest at SCRC-VIXEN: ]
  90:   25 29-Dec DODDS→LispM-Users        New World for the New Year
```

```
Received: from scrc-yukon by scrc-vixen with CHAOS; 29 Dec 1983 14:15:55-EST
Received: from SCRC-SEINE by SCRC-YUKON with CHAOS; Thu 29-Dec-83 14:16:38-EST
Date: Thursday, 29 December 1983, 14:16-EST
From: ??? <Zmailtest at SCRC-VIXEN>
Subject:
To: Zmailtest at SCRC-VIXEN

This is an example.

Message
Zmail VIXEN: /usr2/abc.mailbox   Msg #88/90 (forwarded)  ()
```

Figure 2. Top-level Display with Mail File

`:`	The message has been or is displayed.
`A`	The message has been answered.
`D`	The message has been deleted.

(The above list is in reverse order of precedence; that is, a deleted message is marked `D` whether or not it has been answered.)

`Lines`	The message length in lines.
`Date`	The date the message was sent.
`From→To`	The sender (`From` field) and as much of the recipients (`To` field) of the message as will fit, summarized on either side of the `→`. A missing name before or after the arrow means the message was from or to you. For example, `→PJF,,MJH` represents a message from you to PJF, yourself, and MJH. Only the To: recipients are listed, not the Cc: or Bcc: recipients. See the section "Sending Your Mail", page 62. See the section "Commands for Sending Mail" in *Communicating with Other Users*.
Keywords	The keywords attached to the message are enclosed in braces.
`Subject or Text`	The Subject: field of the message, or in the absence of a Subject: field, the first line of meaningful text in the message.

6.1.2.2 Command Menu

The *Command Menu* provides a mouse-sensitive menu of the most useful top-level commands. Some of these commands (for example, [Delete]) apply only to the current message. In this manual, when we say, for example, "[Get inbox]", we mean the Get inbox command in this menu.

6.1.2.3 Message Window

The *Message Window* displays the current message. The message window is an editor buffer.

Initially, there is no current message; instead, there is a short note explaining how to read and send mail. When you read your mail, the first new message becomes the current message; if there is no new mail, the first old message is the current message. As you move around the mail file to inspect other messages, they are selected as the current message and displayed.

6.1.2.4 Zmail Minibuffer

The minibuffer contains the mode line. It is also where some short notifications get displayed .

Mode Line

The various information included is:

Program status The mode the program is in. Possibilities are:

`Zmail`	Zmail is at top level.
`Zmail Mail`	Zmail is in mail mode, in which mail is sent. Following the word `Mail` is the word `(Text)`, which identifies the mode in which the message to be sent is being edited. The editor mode is followed by either `Message`, `Headers`, or `Mail`, indicating which window the cursor is in. (For a description of these windows, see the explanation of the `c-X 0`, `c-X 1`, and `c-X 2` commands. See the section "Configuring and Selecting Zmail Windows" in *Communicating with Other Users*. See the section "Sending Your Mail", page 62. See the section "Replying to Mail", page 67.
`Zmail Profile`	Zmail is in Profile mode, in which you can customize Zmail. See the section "Customizing Zmail", page 170. Following the word `Profile` is the name of your init file, in which the customizations are stored.
`Zmail Marking`	Zmail is in Marking mode, executing the mark-survey command.
`Zmail Editing Message`	Zmail is in Editing Message mode, in which you can edit your copy of a previously received message.

Current mail file The name of the current mail file, or "`No current mail file`" if there is none.

Current message number/total number of messages

Message properties Properties describing the current message, in parentheses. Possible properties are:

`unseen`	Message is now being seen for the first time
`deleted`	Message has been marked for deletion
`recent`	Message was new mail in the current session
`last`	Message is the last in the file
`filed`	Message has been copied to another file
`answered`	Message has been replied to

`forwarded`	Message has been forwarded
`redistributed`	Message has been redistributed
`badheader`	Message has a bad header

Keywords	Any keywords that have been saved on this message, in braces.

Second Mode Line

The second mode line gives useful information on what the program is doing at various times. In Figure 2, for example, the new mail message means Zmail detected new mail in your inbox. Other messages that appear in the second mode line tell you what file the program is reading or writing, what error just occurred (Zmail flashes the screen also), or what certain keys do (for example, END and ABORT). It is a good idea to check the mode lines if you are unsure where in the program you are or how to get elsewhere.

Most of the screen is mouse-sensitive.

Figure 2 shows Zmail at top level. The information about the current mail file and current message is only displayed at top level; the first mode line is used for other information when the program is in different modes.

6.1.3 Sending Your Mail

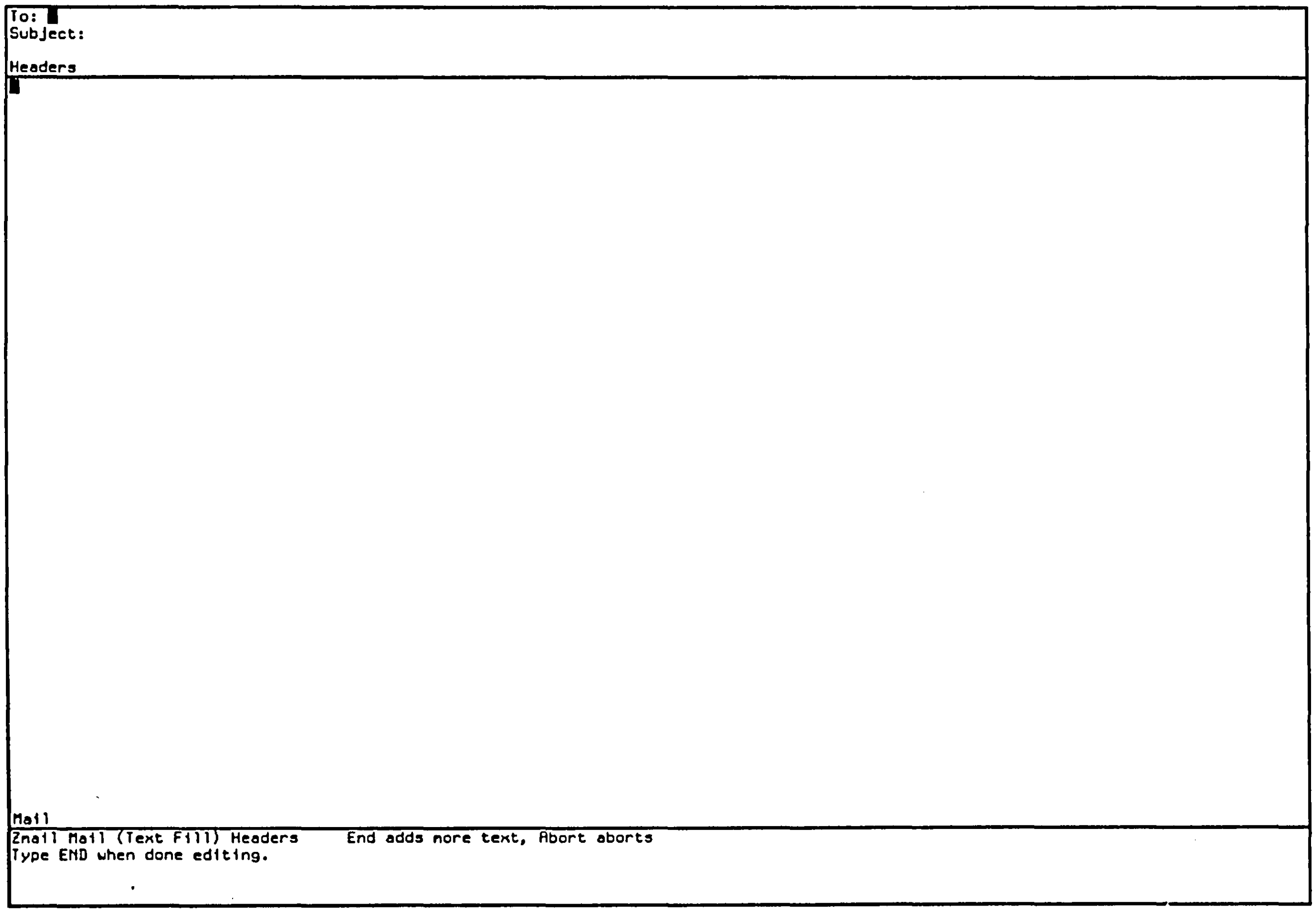
To send mail, use [Mail], which is displayed in the command menu. Zmail displays two windows, one for the message headers, and one for the message itself. (See 3)

At this point, the headers window is selected, with the cursor following the word `To:`. The program is prompting you for the contents of the To: field, which specifies to whom the message is to be sent. Respond by entering a list of one or more user names or mailing lists separated by commas.

If you wish to send someone a carbon copy of the message (which means they also get the message, but are not considered a primary recipient), press RETURN, then type `Cc:` followed by a list of one or more user names or mailing lists, separated by commas. If you want to save a copy of the message for yourself, include your own name on the `Cc:` list (or on the `To:` list).

It is recommended that you include a `Subject:` field with your message, as this is then used in the summary display of the recipient's mail file. (If you have no `Subject:` field, the text of the first meaningful line is used.) To add a `Subject:` field, press RETURN, then type `Subject:` (or `S:` for short), followed by a line giving the subject of your message.

To enter the message itself, select the message window by pressing END. The message window is an editor window; you can type in the message using all the commands of the editor. See the section "Zmacs Manual" in *Text Editing and Processing*. The headers window is also an editor window.



```
To: █
Subject:

Headers
█










Mail
Zmail Mail (Text Fill) Headers     End adds more text, Abort aborts
Type END when done editing.
```

Figure 3. Mail Mode Display (One-window Mode)

At any time during editing you can return to the headers window to add or change entries; just click left on the headers window. To get back to the mail window, press END or click left on the mail window.

If you change your mind while working on the message and decide that you do not want to send anything, press ABORT, and you return to top level; nothing is sent. If you later decide that you did want to send the message after all, use [Continue]. See the section "Continuing Completed or Aborted Zmail Messages" in *Communicating with Other Users.*

When you are satisfied, press END to send the message. If you are in the headers window, press END twice. See 4 for a message about to be sent.

If the message is sent successfully, Zmail displays "Message sent" and returns to top level. If there is a problem, Zmail tells you about it and remains in mail mode. Typical problems are omitting the `To:` field, trying to send mail to a nonexistent user, or mistyping a user name. Correct the error and resend the message by pressing END twice.

6.1.4 Reading Your Mail

To read your mail, use [Get inbox]; Zmail reads in your primary mail file (containing old mail) and any new mail.

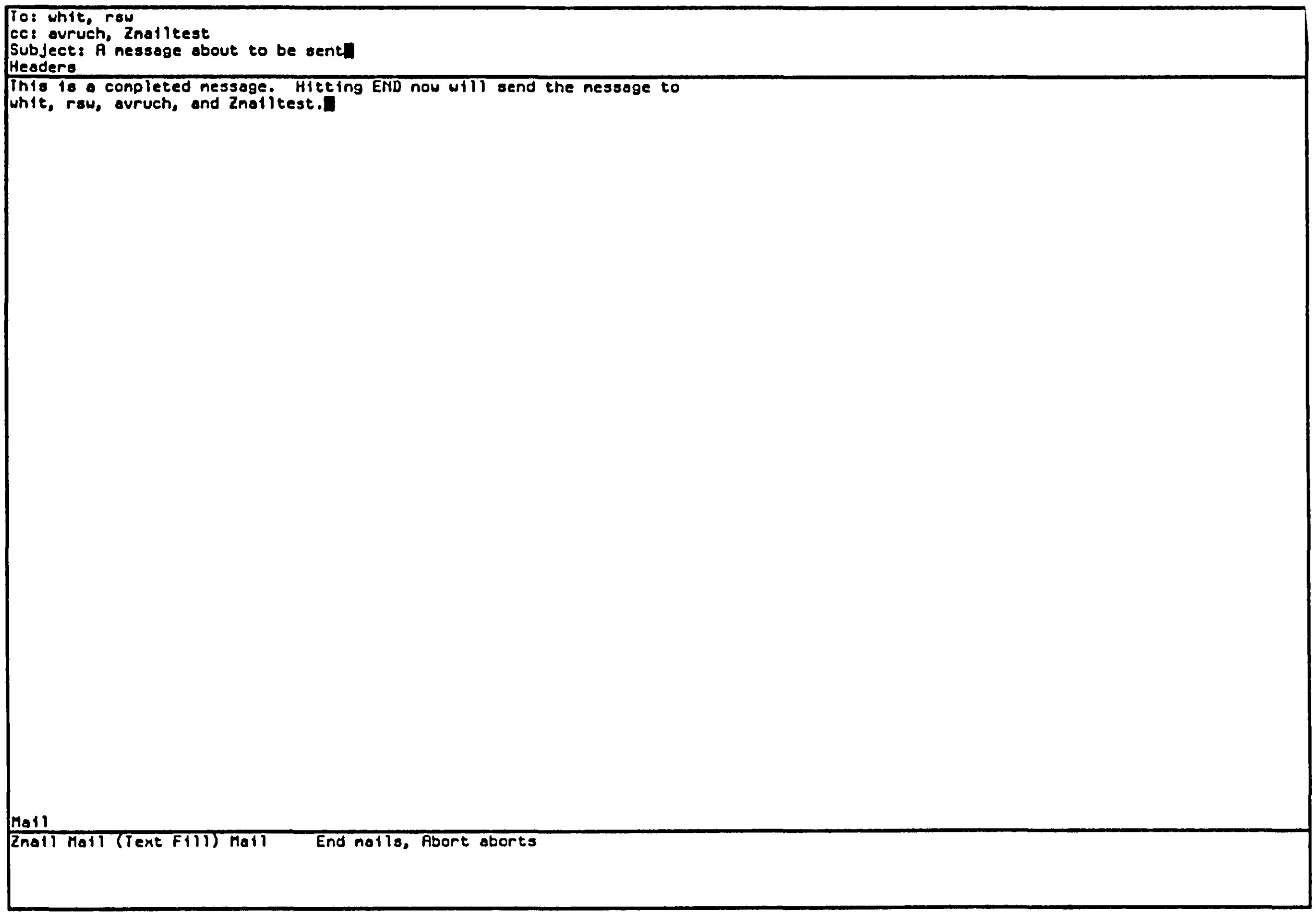
Command	*Meaning*
[Get inbox] or G from the keyboard	Gets the new mail (inbox) for the current buffer. It has no effect when a collection is current.
[Get inbox (M)]	Prompts you for an inbox name for the current buffer.
[Get inbox (R)]	Calls up a menu of possible buffers to get the new mail for.

Two files are involved here: your *primary mail file*, which contains messages you have already seen, and your *inbox*, which contains new mail. If you do not have a mail file — as might be the case the first time you run Zmail — the program offers to create one for you. Press RETURN to let Zmail create the file, or ABORT if for some reason you do not want a mail file. No similar problem with inbox files exists; they are created when needed, and are deleted when Zmail reads your new mail from them.

While an internal data structure used for conversation and reference commands is created, the following message appears in the Zmail minibuffer:

`Creating reference hash table for buffer` *filename*

The parsing required in the creation of reference hash tables is time-consuming for large unparsed files. The appearance of this message in the minibuffer notifies you that it is building a reference hash table so that you do not think something is wrong.



```
To: whit, rsw
cc: avruch, Zmailtest
Subject: A message about to be sent
Headers
This is a completed message.  Hitting END now will send the message to
whit, rsw, avruch, and Zmailtest.
```

```
Mail
Zmail Mail (Text Fill) Mail      End mails, Abort aborts
```

Figure 4. A Message about to be Sent

If you have no new mail, Zmail says so. Otherwise, the summary window starts to scroll as lines appear for new messages, and the first new message is displayed in the message window as the current message.

If the message does not fit entirely in the window, the words --more below-- appear in the mode line. When text is off-screen both above and below, the message reads --more above and below--; when you reach the final screen of the message, the message reads --more above--. Use the following commands to move from screen to screen:

Command	*Action*
SPACE or c-V	Displays the next screen of the message
BACKSPACE or m-V	Goes back to the previous screen
	Returns to the beginning of the current message

To use the mouse for scrolling, you can click left on the --more-- message to scroll forward one screen, or click middle to scroll back one screen. If you click right, you get a menu of four items: [Forward] and [Backward], which move forward and backward by one screen, and [Beginning] and [End], which move to the beginning and the end of the message. For more precise control of scrolling, use the scroll bar in the left margin of the window. See the section "Scrolling".

6.1.5 What to Do After Reading a Message

Once you have finished reading a particular message, there are several things you might want to do. You might want to read the next new message (if any), you might want to delete the message if it is no longer of value, or you might want to reply to the message.

6.1.5.1 Deleting and Undeleting Messages

After you have finished reading a message, you often want to delete it and move on to the next one. To do this, use [Delete]. This marks the message as deleted — a D appears in its summary line — and moves to the next message.

If you change your mind, you can undelete a message; use [Undelete]. This starts at the current message and searches backward for a deleted message, undeletes it, and selects it as the current message. Deleted messages do not actually disappear until the mail file is saved, which is why Undelete is possible.

6.1.5.2 Moving Among Messages

When you finish reading a message that you do not want to delete, use [Next] to read the next message. To go back to the previous message, use [Previous]. To jump to the first message in the file, use [Previous (M)]; for the last message, use [Next (M)]. (Note: These commands ignore deleted messages; they actually give you the next undeleted message, previous nondeleted, first nondeleted, and last undeleted.)

To read an arbitrary message, select it from the summary window by clicking left on its summary line. If the summary does not all fit in the window, you might first have to scroll the display using the left-margin scroll bar.

6.1.5.3 Replying to Mail

To reply to the current message, use [Reply] or [Reply (M)]. This sets up the screen as three windows: the Message window displays the current message, the Headers window contains the reply headers, and the Mail window is where you write the reply itself. (See Figure 5.)

The cursor is in the Mail window, so you can just type in the text of the message, using editor commands to edit what you are typing. To send the message, press END. If you change your mind and do not want to reply, press ABORT. If you want to edit the headers, you can select the Headers window by clicking left on it. These commands are the same as in mail mode. See the section "Sending Your Mail", page 62.

What is special about reply mode is that the reply headers are written automatically. The headers that Zmail writes are the `To:` field, the `CC:` field, the `Subject:` field, and the `In-Reply-To:` field. The `Subject:` field is simply a copy of the original `Subject:`. Defaults for the `To:` and `CC:` fields are provided. Notice the mouse-documentation line. To set up alternate `To:` and `CC:` fields, use [Reply (R)] and choosing from the pop-up menu the combination of `To:` and `CC:` you want.

6.1.5.4 Saving the Mail File

When you have finished reading your new mail, you should save your mail file by using [Save]. This expunges deleted messages from the file and then saves it, writing the modified mail file back out to the file system where it is kept until next time.

If you now wish to leave Zmail, select another program using the SELECT key or the System menu.

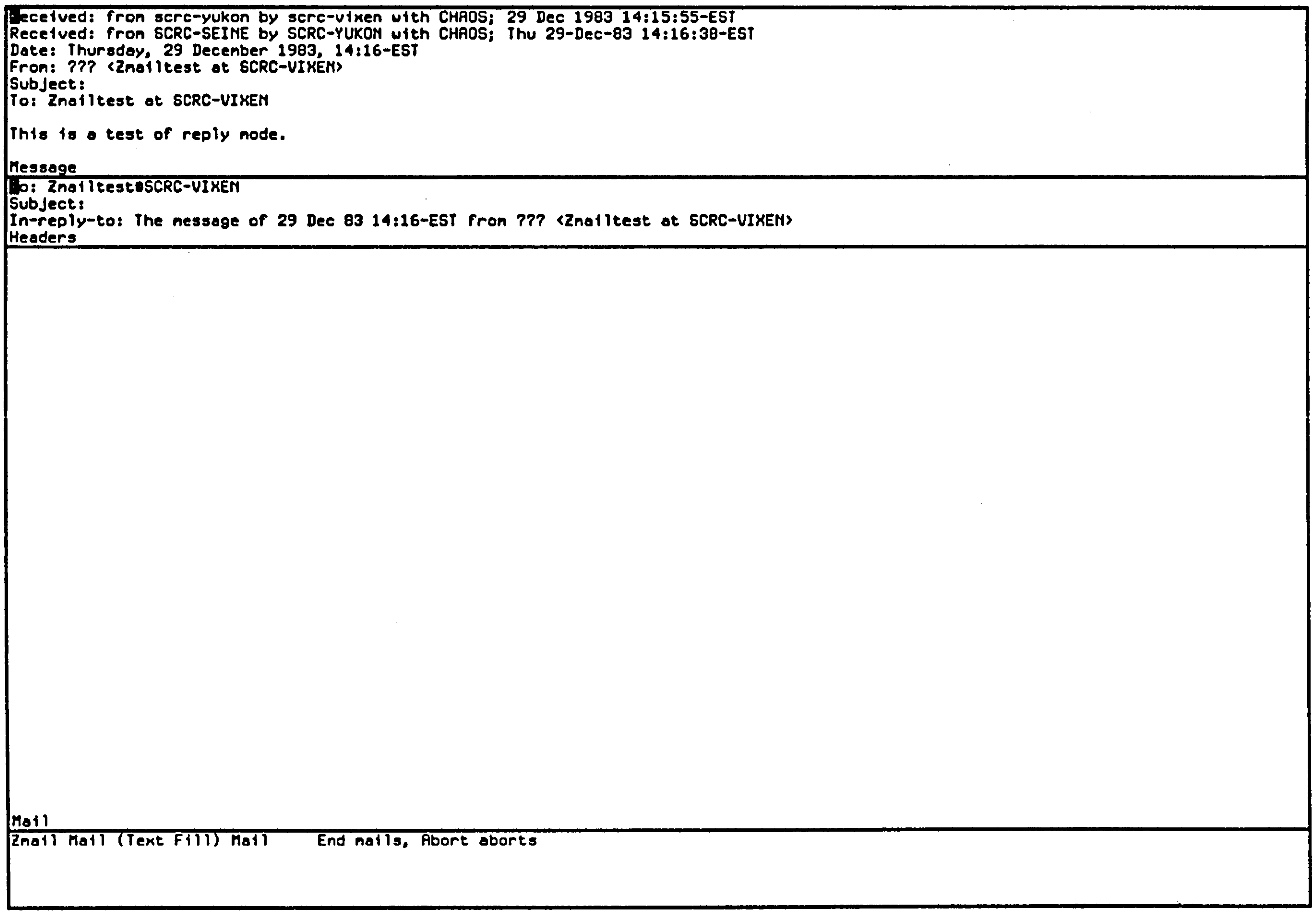
6.1.6 Getting Fancy with Zmail

6.1.6.1 Encrypting Messages

Zmail supports encryption. Commands are available both when you are composing mail and when you are reading mail. Encrypted messages contain a new header field to indicate that they contain encrypted text.

The command to encrypt a message draft is Encrypt Text (m-X). Use it after you have completed the message draft but before you send it. Zmail prompts for an encryption key that the recipient must provide in order to decrypt the message. It converts the draft to a form that you cannot read. Decrypt Text is also available for message drafts. Both of these commands appear on the draft editor menu.

Decrypt Msg (m-X) displays an encrypted message as plain text, prompting for the



```
Received: from scrc-yukon by scrc-vixen with CHAOS; 29 Dec 1983 14:15:55-EST
Received: from SCRC-SEINE by SCRC-YUKON with CHAOS; Thu 29-Dec-83 14:16:38-EST
Date: Thursday, 29 December 1983, 14:16-EST
From: ??? <Zmailtest at SCRC-VIXEN>
Subject:
To: Zmailtest at SCRC-VIXEN

This is a test of reply mode.

Message
To: Zmailtest@SCRC-VIXEN
Subject:
In-reply-to: The message of 29 Dec 83 14:16-EST from ??? <Zmailtest at SCRC-VIXEN>
Headers

Mail
Zmail Mail (Text Fill) Mail      End mails, Abort aborts
```

Figure 5. Mail Mode Display (Two-window Mode)

encryption key. By this operation, you are only viewing the plain text form; use a numeric argument to store the plain text version in the mail file.

Text yanked by Forward and Reply prompts for a decryption key rather than yanking unreadable text.

The only encryption algorithm currently supported is the NBS algorithm, used by Hermes.

6.1.6.2 Fonts in Messages

Zmail can interpret messages that contain fonts. These are the same fonts that the editor uses in Set Fonts (m-X) and the editor font commands.

6.2 Talking to Other Users

6.2.1 Introduction to Converse

Converse is a facility for communicating interactively with other logged-in users.

The Converse interactive message editor is operated by a window with its own process. Converse keeps track of all of the messages that you have received or sent. The Converse window shows all of the messages that have been sent or received since the machine was cold booted.

Messages sent between you and another user are organized into a *conversation*. Conversations are separated from each other by a thick black line. Within each conversation are all messages, outgoing and incoming, arranged in chronological order, and separated by thin black lines.

You can use Converse to look at conversations, send messages, and receive messages. Converse is built on the Zwei editor, so you can edit your message as you type it in, or pick up and move around text between one message and another, or among messages, files, and pieces of mail.

To enter Converse, do one of the following:

- Press SELECT C.
- Evaluate **(qsend)**.
- Use [Select / Converse] in the System menu.
- Answer C in the Converse pop-up window when a message arrives.

6.2.2 Using Converse

When you enter Converse for the first time, the window is empty except for a blank message at the top of the screen, starting with To:. You start a message by filling in a recipient after the To: and typing the message text. To send the message, press END. When the message has been sent successfully, it appears as a conversation. A blank message remains at the top of the screen, and just below that a heavy black line delimits the message(s) of the conversation you just started. Just below the heavy black line is another blank message, but this one has the name of the other person in the conversation filled in. Below this blank message, separated by a thin black line, appears the message you just sent, with the date and time it was sent.

When the person to whom you sent the message replies, the reply appears in the conversation above the message you sent, and below the blank message. Your cursor is left in the blank message so you can reply easily.

You can use the regular editor commands to move around in the Converse window. There are two commands specific to Converse that are particularly useful: c-m-] (move to next conversation) and c-m-[(move to previous conversation).

You exit from Converse by pressing ABORT or by selecting another window. You can also press c-END when sending a message to send the message and exit from Converse.

To start a conversation, enter Converse, go to the top of the Converse window and fill in the blank message, starting with the To: line to specify the new recipient. Finish by pressing END to send the message. To send the message and exit Converse, finish by pressing c-END.

To send a message as part of an existing conversation, find that conversation in Converse and fill in the blank message at the beginning of the conversation, finishing by pressing END to send the message, or by pressing c-END to send the message and exit Converse.

You do not have to be in the main Converse window to receive messages. Converse will deliver a message to you in any window. Since this might be annoying, you can customize what happens when a message arrives by using the variable **zwei:*converse-mode***. See the section "Customizing Converse", 172.

When you are in a window other than Converse and a new message arrives, a window pops up at the top of the screen displaying the message. You can respond R to type in a reply, N (for "no action") to make the message window deexpose, or C to enter Converse. Entering Converse has several advantages: you can look over the previous messages in the conversation, and you can use the editor to help you construct a reply.

Converse remembers all messages that you send or receive, even if you did not use the main Converse window to send them or reply to them.

Converse lets you know as soon as a message comes in, by beeping or flashing the screen, and if it is supposed to notify you, it does so without waiting for the main Converse process to wake up. In pop-up mode, if the pop-up message window is already in use, an incoming message causes the message window to beep or flash but not to display the message. This is necessary since only one message at a time should pop up. When the message window is deexposed it is reexposed immediately with the new message in it.

If the main Converse window is exposed, a new message is shown there with its conversation; it is never shown via a notification or a pop-up message window. If the main Converse window is exposed but its process is busy (typically, when it is in the Debugger or in an editor command and waiting for typein), Converse beeps or flashes but does not display the message. You can display the message by clearing the Converse process. You can usually clear the Converse process by pressing ABORT.

6.2.2.1 Converse Commands

Converse has several commands for managing your conversations.

HELP	Displays a summary of Converse commands.
END	Sends the current message. The behavior of this key can be changed by the variable **zwei:*converse-end-exits***.
c-END	Sends the current message and exits from Converse. The behavior of this key can be changed by the variable **zwei:*converse-end-exits***.
ABORT	Exits Converse.
c-M	Mails the current message instead of sending it. This is useful if Converse reports that the person you want to send the message to is not logged in anywhere.
c-m-[	Moves to the previous conversation.
c-m-]	Moves to the next conversation.
m-X Delete Conversation	Deletes the current conversation from the Converse window.
m-X Write Buffer	Writes the entire Converse buffer (all conversations) to a file. It prompts for a pathname.
m-X Write Conversation	Writes only the current conversation to a file. It prompts for a pathname.
m-X Append Buffer	Appends the entire Converse buffer (all conversations) to the end of a file. It prompts for a pathname.

m-X Append Conversation

Appends only the current conversation to the end of a file. It prompts for a pathname.

m-X Regenerate Buffer

Rebuilds the structure of the Converse buffer. This might be necessary if you damage the buffer in some way, for instance by removing one of the black lines separating conversations. Some error messages might ask you to give this command and try again. The message you are currently typing might be lost, but you can prevent this by putting the text on the kill ring before issuing the m-X Regenerate Buffer command.

6.2.2.2 Lisp Listener Commands for Converse

zwei:qsends-off &optional (*gag-message* **t**) *Function*

Sometimes, you might wish not to be interrupted with interactive messages. A function called **zwei:qsends-off** exists for such occasions. If you give it a string argument, *gag-message*, the variable **zwei:*converse-gagged*** is set to this string and the string is returned to anyone who tries to send a message to you. Otherwise, they just get a note saying that you are not accepting messages. **zwei:qsends-on** toggles **zwei:*converse-gagged***.

zwei:qsends-on *Function*

After using **zwei:qsends-off** to notify interactive message senders that you are not accepting messages, **zwei:qsends-on** allows interactive messages to be received again.

chaos:notify-local-lispms &optional (*message* **(zwei:qsend-get-message "all lisp machines")**) *Function*

Sends *message* to all Lisp Machines at your site based upon information it gets from the namespace database about the Lisp Machines at the local site. *message* should be a string; if it is not provided, the function prompts for a message. Each recipient receives the message as a notification, rather than as an interactive message.

qsend &optional *destination message* *Function*

Sends interactive messages to users on other machines on the network.

destination is normally a string of the form "*name@host*", to specify the recipient. If you omit the *@host* part and just give a name, **qsend** looks at all of the Lisp Machines at your site to find any that *name* is logged into; if the user is logged into one Lisp Machine, it is used as the host; if more than one, **qsend** asks you which one you mean. If you leave out *destination* altogether, doing just **(qsend)**, Converse is selected as if you had pressed SELECT C.

message should be a string. If it is omitted, **qsend** asks you to type in a message. You should type in the contents of your message and press END when you are done.

The input editor is used while you type in a message to **qsend**. So you get some editing power, although not as much as with full Converse (since the latter uses Zwei). See the section "Using the Input Editor". This function predates Converse and is retained for compatibility.

print-sends &optional (*stream* **standard-output**) *Function*

Prints out all messages you have received (but not messages you have sent), in forward chronological order, to *stream*. Converse is more useful for looking at your messages, but this function predates Converse and is retained for compatibility.

qreply &optional *text* *Function*

Sends a reply to the Converse message received most recently. You can supply a string as the text of the message or omit it and let **qreply** prompt for it. It returns a string of the form "*user@host*", indicating the recipient of the message. This function predates Converse and is retained for compatibility.

7. How to Change Fonts

7.1 What Are Fonts?

On the Symbolics Lisp Machine, characters can be typed out in any of a number of different typefaces. Some text is printed in characters that are small or large, boldface or italic, or in different styles altogether. Each such typeface is called a *font*. A font is conceptually an array, indexed by character code, of pictures showing how each character should be drawn on the screen. The Font Editor (FED) is a program that allows you to create, modify, and extend fonts.

A font is represented inside the Lisp Machine as a Lisp object. Each font has a name. The name of a font is a symbol, usually in the **fonts** package, and the symbol is bound to the font. A typical font name is **tr8**. In the initial Lisp environment, the symbol **fonts:tr8** is bound to a font object whose printed representation is something like:

```
#<font tr8 234712342>
```

The initial Lisp environment includes many fonts. Usually there are more fonts stored in BFD files in file computers. New fonts can be created, saved in BFD files, and loaded into the Lisp environment; they can also simply be created inside the environment.

7.2 Displaying Fonts

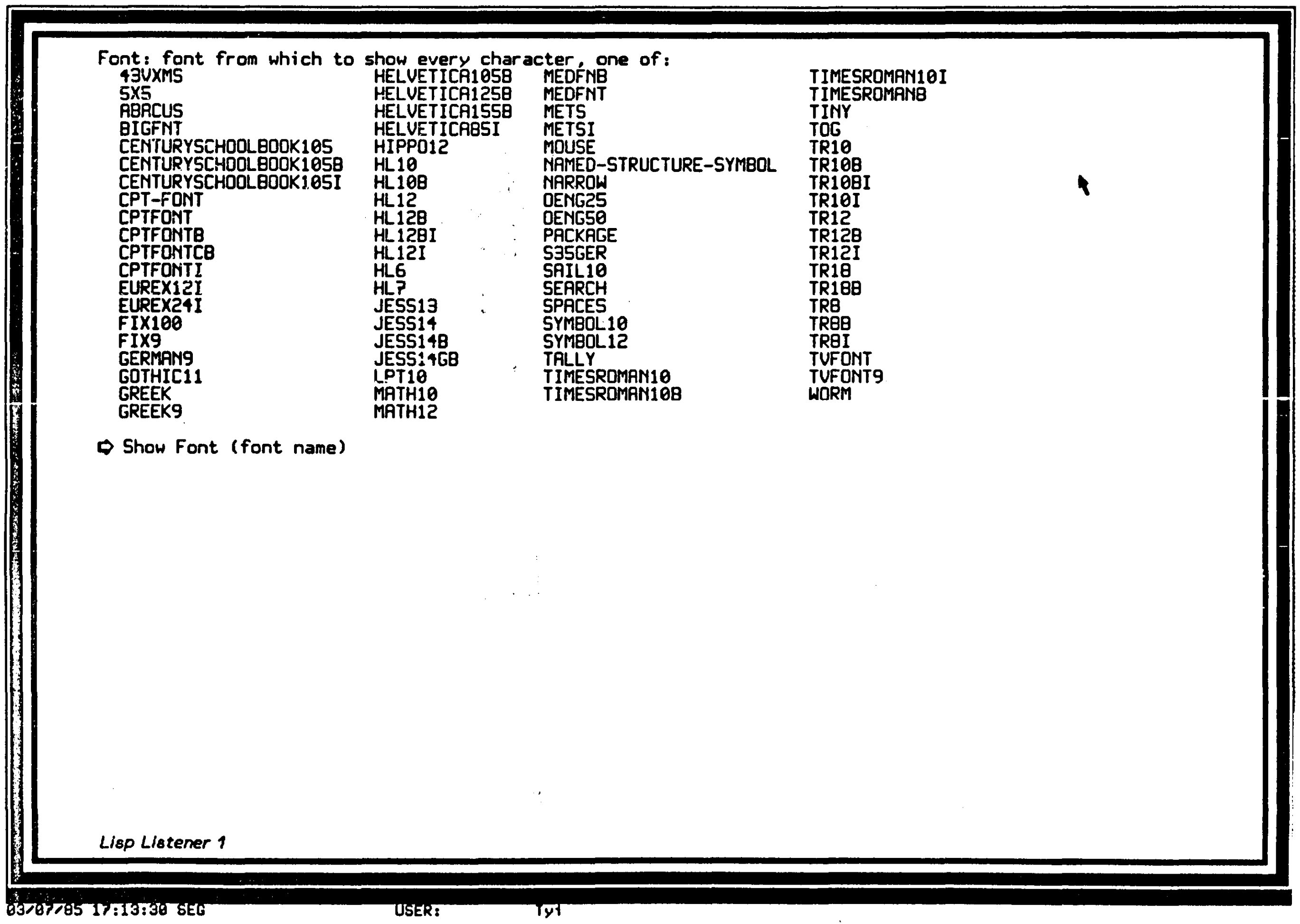
The Show Font command displays fonts. The Show Font command has no default—it displays either:

- The names of all the fonts that have been automatically loaded into the system (Show Font SPACE HELP)

- If you specify one of those names, all characters as they appear in the specified font (Show Font *font-name*)

7.3 Standard Lisp Machine Fonts

You can use `Show Font HELP` in the Lisp Listener or the List Fonts (m-X) command in Zmacs to get a list of all the fonts that are currently loaded into the Lisp environment. The **fonts** package contains the names of all fonts. Here is a list of some of the useful fonts:

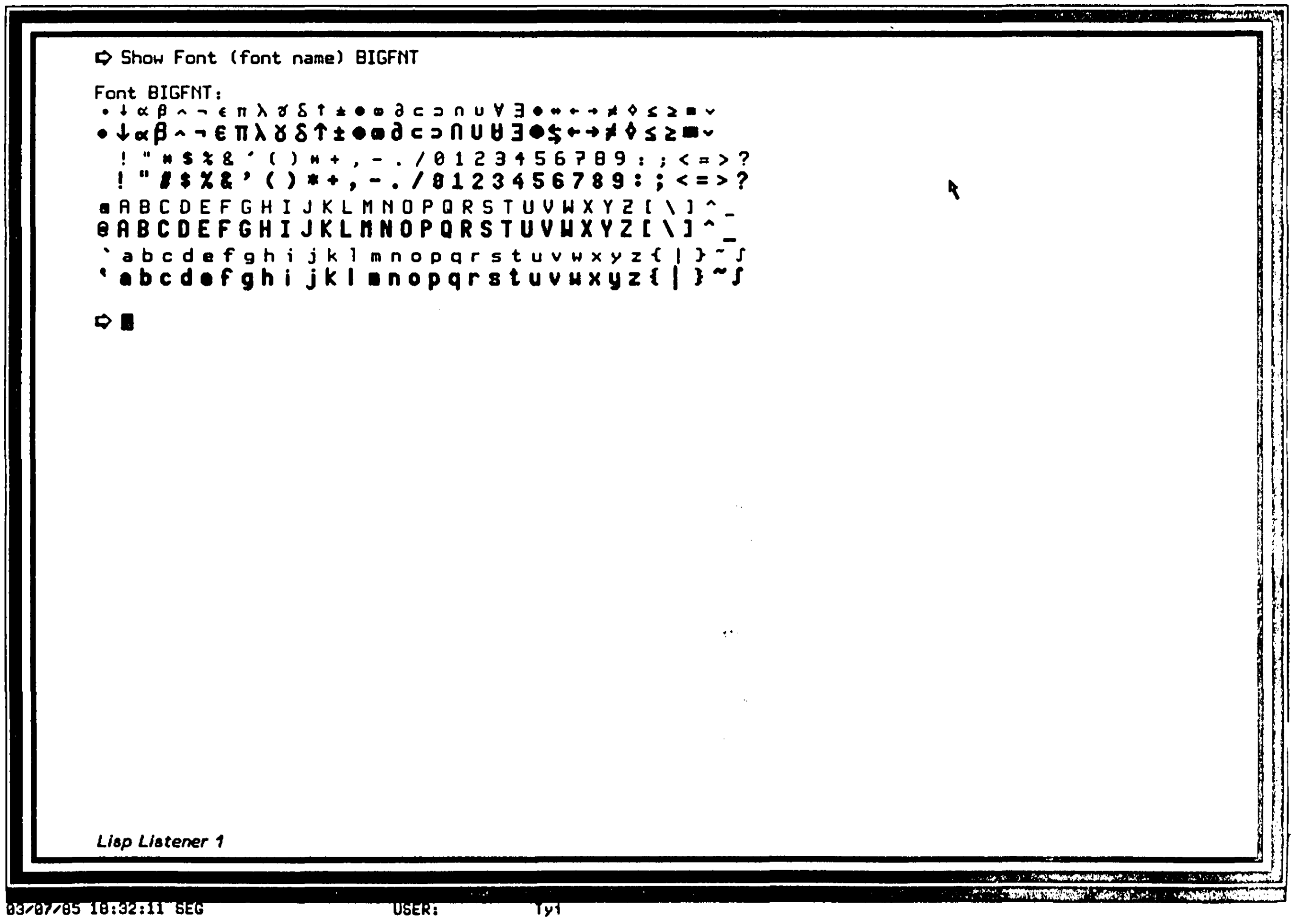


```
Font: font from which to show every character, one of:
  43VXMS                   HELVETICA105B   MEDFNB                  TIMESROMAN10I
  5X5                      HELVETICA125B   MEDFNT                  TIMESROMAN8
  ABACUS                   HELVETICA155B   METS                    TINY
  BIGFNT                   HELVETICA85I    METSI                   TOG
  CENTURYSCHOOLBOOK105     HIPPO12         MOUSE                   TR10
  CENTURYSCHOOLBOOK105B    HL10            NAMED-STRUCTURE-SYMBOL  TR10B
  CENTURYSCHOOLBOOK105I    HL10B           NARROW                  TR10BI
  CPT-FONT                 HL12            OENG25                  TR10I
  CPTFONT                  HL12B           OENG50                  TR12
  CPTFONTB                 HL12BI          PACKAGE                 TR12B
  CPTFONTCB                HL12I           S35GER                  TR12I
  CPTFONTI                 HL6             SAIL10                  TR18
  EUREX12I                 HL7             SEARCH                  TR18B
  EUREX24I                 JESS13          SPACES                  TR8
  FIX100                   JESS14          SYMBOL10                TR8B
  FIX9                     JESS14B         SYMBOL12                TR8I
  GERMAN9                  JESS14GB        TALLY                   TVFONT
  GOTHIC11                 LPT10           TIMESROMAN10            TVFONT9
  GREEK                    MATH10          TIMESROMAN10B           WORM
  GREEK9                   MATH12

⇨ Show Font (font name)

Lisp Listener 1
03/07/85 17:13:30 SEG                USER:         Tyi
```

Figure 6. Show Font Display of Fonts



```
➪ Show Font (font name) BIGFNT

Font BIGFNT:
• ↓ α β ^ ¬ ε π λ γ δ ↑ ± ● ∞ ∂ ⊂ ⊃ ∩ ∪ ∀ ∃ ● ↔ ← → ≠ ◊ ≤ ≥ ■ ∨
• ↓ α β ^ ¬ ε π λ γ δ ↑ ± ● ∞ ∂ ⊂ ⊃ ∩ ∪ ∀ ∃ ● $ ← → ≠ ◊ ≤ ≥ ■ ∨
  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ ∫
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ ∫

➪ ▮

Lisp Listener 1
03/07/85 18:32:11 SEG          USER:      Ty1
```

Figure 7. Show Font Display of Font Characters

fonts:cptfont	This is the default font, used for almost everything.
fonts:jess14	This is the default font in menus. It is a variable-width rounded font, slightly larger and more attractive than medfnt.
fonts:cptfonti	This is a fixed-width italic font of the same width and shape as **fonts:cptfont**, the default screen font. It is most useful for italicizing running text along with **fonts:cptfont**.
fonts:cptfontcb	This is a fixed-width bold font of the same width and shape as **fonts:cptfont**, the default screen font.
fonts:medfnt	This is a fixed-width font with characters somewhat larger than those of **cptfont**.
fonts:medfnb	This is a bold version of **medfnt**. When you use Split Screen, for example, the [Do It] and [Abort] items are in this font.
fonts:hl12i	This is a variable-width italic font. It is useful for italic items in menus; Zmail uses it for this in several menus.
fonts:tr10i	This is a very small italic font. It is the one used by the Inspector to say *"More above"* and *"More below"*.
fonts:hl10	This is a very small font used for nonselected items in Choose Variable Values windows.
fonts:hl10b	This is a bold version of **hl10**, used for selected items in Choose Variable Values windows.

7.4 Setting Fonts

7.4.1 Setting Fonts in the Input Editor

The *font map* contains a list of all fonts available to the input editor and the command processor; these programs display typein in one of the fonts given in the map. You can explicitly set the font map, that is, name the fonts that you want to use, with the m-J command, or you can use the default contents of the font map. When you log in, the font map always contains (at least) one font.

The font you are currently using is known as the *default font*. To use a different font in the font map you must explicitly *select* it with the c-J command.

When you set the map (name the fonts), the input editor assigns a number to each font in the order in which you type the names. The first font named is font #0, the second font, #1, and so on. When you select a font from the list, you refer to it by its number, not its name. The input editor then translates the number to the name of the font. For example, suppose you specify the names of fonts in your font map to be **cptfont** and **bigfnt**, in that order; the input editor maps the number #0 to **cptfont** and the number 1 to **bigfnt**. Unless you reset the map, font #0 means **cptfont**. Font #0 is also the default font, unless you subsequently select another font in the map as the default. For example, to have typein displayed in **bigfnt**, select this font from the map by typing its number and the c-J command: c-1 c-J. Note that selecting a font does not change its placement in the list.

If the font you want to use is not named in the font map, you must first reset the map to make that font available to the input editor. To reset the map you must retype the entire font map. For example, suppose, in addition to **cptfont** and **bigfnt**, you want to make **medfnt** available to the input editor. Type:

```
m-J cptfont bigfnt medfnt RETURN
```

To select **medfnt**, type: c-2 c-J.

7.4.1.1 Input Editor Font Commands

m-J: Sets the font map in the current Lisp Listener. It prompts for the names of one or more fonts, separated by spaces.

m-HELP: Displays the names of the fonts in the font map and other state information relating to the input editor.

c-J: Sets the typein font number in the current Lisp Listener. Preceded by a numeric argument, this command sets the font name to be that which corresponds to that numeric argument in the font map.

7.4.1.2 Example

To set the fonts:

```
m-J cptfont bigfnt RETURN
```

Once you have set the fonts, to set the default typein font to be **cptfont** type:

```
c-0 c-J
```

To set it to **bigfnt** type:

```
c-1 c-J
```

c-J without any numeric argument is the same as c-0 c-J.

7.4.2 Setting Fonts in Zmacs

Zmacs sets fonts in two ways:

- It reads them from attribute list of the file or buffer
- It allows you to explicitly specify them

In addition, Zmacs allows you to specify different fonts for individual characters, words, and regions.

7.4.2.1 In the Attribute List

The attribute list at the top of your buffer or file specifies information about the attributes of the file, including font information:

```
-*- Mode: Fundamental; Fonts: JESS14,BIGFNT; -*-
```

(See the section "Buffer and File Attributes in Zmacs" in *Text Editing and Processing*.) If the attribute list appears in a file, Zmacs binds the attributes it specifies to the values in the attribute list when you read or load the file. In the case of fonts, it sets any fonts listed as the current fonts, mapping them to the code letters and displaying the text in the buffer correspondingly. For example, when reading in a file containing the above attribute list, Zmacs maps **jess14** to font A, **bigfnt** to font B, and displays the text accordingly. References to the default font now refer to A, which is **jess14**, in which Zmacs displays subsequent typein.

To set the fonts in the attribute list, you can manually edit it, use the Set Fonts (m-X) command, or use the attribute-manipulating commands Update Attribute List (m-X) and Reparse Attribute List (m-X). These three commands immediately update the attribute list and display font changes in the buffer. See the section "Update Attribute List" in *Text Editing and Processing*. See the section "Reparse Attribute List" in *Text Editing and Processing*.

7.4.2.2 With Set Fonts (m-X)

The font you are currently using is known as the *default font*. To use different fonts in Zmacs, you must first name (set) them so that Zmacs knows about them. You can explicitly set the fonts that you want to use with the Set Fonts (m-X) command. Then you can select one of these named fonts with c-m-J.

When you set the fonts, Zmacs assigns a letter to each font in the order in which you type the names. The first font named is font A, the second font, B, and so on. When you select a font from the list, you refer to it by its letter, not its name. Zmacs then translates the letter to the name of the font. For example, suppose you specify the names of fonts in Zmacs to be **cptfont** and **bigfnt**, in that order; the input editor maps the letter A to **cptfont** and the letter B to **bigfnt**. Unless you reset the fonts, font A means **cptfont**. Font A is also the default font, unless you subsequently select another font as the default. For example, to have subsequent typein displayed in **bigfnt**, select this font from the map by typing the c-m-J command and then B.

If you have not previously set the font you want to use, you must first set the fonts to make that font available to Zmacs. To reset the fonts you must retype the entire list of fonts. For example, suppose, in addition to **cptfont** and **bigfnt**, you want to make **medfnt** available to Zmacs. Type:

```
m-X Set Fonts RETURN
cptfont bigfnt medfnt RETURN
```

To select **medfnt**, type: c-m-J and then C.

7.4.2.3 Zmacs Font Commands

When one of the Zmacs font commands prompts you for the name of a font, you can:

- Type a font letter
- Press ESCAPE to enter a new font name in a minibuffer
- ([L]) on any character selects its font
- ([R]) displays a menu of loaded fonts

Display Font (m-X), Show Font (m-X): Prompts for the name of a font (`Font to display:`).

List Fonts (m-X): Lists the fonts that have been automatically loaded in your world. With a numeric argument, it also lists the font files on the file computer. It offers to display any of the fonts listed (`Click on name to display a sample`).

Set Fonts (m-X): Changes the set of fonts to use. It reads a list of font names, separated by spaces, commas, or both, from the minibuffer and also offers to clear all previously set (`fonts font1, font2, ...: (Return to clear fonts)`).

c-m-J: Changes the default font. It prompts for the name (letter) of the new font in the echo area. The size of the blinker adjusts correspondingly to the size of the font characters. It does not redisplay the buffer in the newly specified font.

c-J: Changes the font of one or more characters forward. It prompts for the name (letter) of the new font in the echo area. With a numeric argument *n*, it changes the font for the next *n* characters. It immediately displays the character in the newly specified font.

m-J: Changes the font of one or more words forward. It prompts for the name (letter) of the new font in the echo area. With a numeric argument *n*, it changes the font for the next *n* words. It immediately displays the word in the newly specified font.

c-X c-J: Changes the font of the text in the region. It prompts for the name (letter) of the new font in the echo area. It immediately displays the region in the newly specified font.

8. Getting Help

The Symbolics-Lisp environment contains many help facilities. This chapter summarizes the facilities for finding out information about the program you are writing and about the general state of your Lisp environment.

This chapter is a collection of the support tools and facilities available for finding the kind of information that you need while programming. It is not exhaustive but suggestive. It does not recommend strategies for applying these facilities but rather lays out what is available for creating a personal style of using the Symbolics computer effectively.

8.1 Reference Material

See the section "Using the Online Documentation System", page 119.

8.2 HELP Key

The key labelled HELP looks up context-dependent documentation.

HELP	Shows documentation available for the current activity. In some programs, c-HELP, m-HELP, and so on, provide additional documentation.
c-HELP	Shows a list of input editor commands (when typed at a Lisp Listener).
sy-HELP	Shows a list of the special function keys and the special character keys.
SELECT HELP	Shows programs and utilities that you can select using the SELECT key.
FUNCTION HELP	Shows a list of useful functions that you can invoke using the FUNCTION key.

See the section "HELP Key in Any Zmacs Editing Window".

8.3 Interaction with Completion and Typeout Windows

The Symbolics-Lisp software has some general interaction conventions. For example, many editor commands offer name completion. You can apply these facilities to exploring the command space of the machine. This section describes some general facilities and strategies for making more effective use of the machine.

8.3.1 Zmacs Completion

Zmacs minibuffer commands offer *completion*, a facility for reducing the number of keys you need to type to specify a name. As soon as you have typed enough characters for a name to be recognized as unique, you can ask for completion. Up until then, you can ask to see which names are possible completions of what you have typed. You can tell when completion is available; the notation "(Completion)" appears at the right end of the minibuffer label line.

8.3.1.1 Completion for Extended Commands (m-X Commands)

The following table summarizes the keys that control completion for entering extended commands.

Key	*Action in m-X commands*
SPACE	Completes the words up to the current word, as far as they are unique.
HELP or c-?	Shows the possible completions in the typeout area.
(R)	Pops up a menu of the possible completions.
c-/	Runs Apropos for each of the partially typed words in the name.
COMPLETE	Displays the full command name, if possible.
RETURN, END	Confirms the command when possible, whether or not you have seen its full name.

Request completion by typing either COMPLETE or RETURN. Using COMPLETE shows the completed name, requiring a further RETURN to confirm it; using RETURN gets you completion and confirmation in one step.

Any time you are typing in a Zmacs extended command name, completion is available. Zmacs command name completion works on initial substrings of each word in the command. For example, "m-X e z" is enough to specify the extended command "Edit Zmacs Command".

Until Zmacs can recognize the name as unique, your request for completion just completes as far as possible. Using COMPLETE at this point moves the input cursor to the first ambiguous place in the command name.

Whenever you are entering a name in a minibuffer that offers completion, you can find out all possible completions of what you have typed so far. Two styles are possible. Using HELP or c-? shows the list of completions in the typeout area; the names are mouse sensitive. Using [(R)] shows the list in a pop-up menu. One good strategy for browsing is to look at the list of completions for initial substrings that are common command verbs, like "list" or "set".

8.3.1.2 Completion for m-.

The m-. (Edit Definition) command offers completion over the set of names that is in the files that have already been loaded into editor buffers. In this case, you request completion with COMPLETE and then confirm it with RETURN.

m-. offers initial substring name completion, with hyphens rather than spaces delimiting the words. For example, "e-d-i" would be sufficient for specifying **edit-definition-internal** (assuming that Zmacs had previously parsed the source file containing it into a buffer).

8.3.2 Completion in Other Contexts

Completion is available in several other contexts, for example, buffer names and package names. Be on the lookout for the presence of "(Completion)" in the minibuffer label line. The conventions for extended commands usually apply.

8.3.3 Typeout Windows in Zmacs

Most of the Zmacs commands for looking up information display the information in a *typeout window*. A typeout window overlays the current buffer display with its contents and disappears as soon as you type any character. Most typeout windows contain mouse-sensitive items. In particular, Zmacs commands and Lisp function specs are mouse sensitive and small menus of operations on the names are available (Arglist, Edit Definition, and so on). See the mouse documentation line.

8.3.4 FEP Command Completion

While the keyboard is connected to the FEP, the following forms of completion are available:

- Pressing the HELP key at the FEP prompt (Fep>) or after typing part of the first word of a command shows the commands understood by the FEP command processor.

- Pressing the HELP key after typing the first word of a command shows a list of commands that begin with that word. Example: set SPACE HELP gives a list of commands that begin with the word set.

8.4 Summary of Help Functions in Different Contexts

Both Zmacs and Lisp offer facilities for finding information either about themselves or about the current environment. In addition, Zmacs offers ways to find information about Lisp functions and variables.

This section lists the names of the functions and commands that are available, grouped according to the context in which they are available. The purpose of this section is to summarize the capabilities and to help you determine both the overall contexts for which you can find help and a particular function that might be what you are looking for. Explanations for each of these functions appear in an alphabetical listing in the third part of this document.

8.4.1 Zmacs Commands for Finding Out About the State of Buffers

Edit Buffers (m-X)
Edit Changed Definitions (m-X)
Edit Changed Definitions Of Buffer (m-X)
List Buffers (c-X c-B)
List Changed Definitions (m-X)
List Changed Definitions Of Buffer (m-X)
List Definitions (m-X)
List Matching Lines (m-X)
Print Modifications (m-X)
Select System as Tag Table (m-X)
Tags Search (m-X)

8.4.2 Zmacs Commands for Finding Out About the State of Zmacs

Apropos (HELP A, m-X)
Describe Variable (m-X)
Edit Zmacs Command (m-X)
List Commands (m-X)
List Registers (m-X)
List Some Word Abbrevs (m-X)
List Tag Tables (m-X)
List Variables (m-X)
List Word Abbrevs (m-X)

8.4.3 Zmacs Commands for Finding Out About Lisp

Describe Variable At Point (c-sh-V)
Edit Callers (m-X)
Edit Definition (m-.)
Edit File Warnings (m-X)

Function Apropos (m-X)
List Callers (m-X)
List Matching Symbols (m-X)
Long Documentation (c-sh-D)
Multiple Edit Callers (m-X)
Multiple List Callers (m-X)
Quick Arglist (c-sh-A)
Show Documentation (m-sh-D)
Show Documentation Function (m-sh-A)
Show Documentation Variable (m-sh-V)
Where Is Symbol (m-X)

8.4.4 Zmacs Commands for Finding Out About Flavors

Describe Flavor (m-X)
Show Documentation Flavor (m-sh-F)
Edit Combined Methods (m-X)
Edit Methods (m-X)
List Combined Methods (m-X)
List Methods (m-X)

8.4.5 Zmacs Commands for Interacting with Lisp

Break (SUSPEND)
Compile And Exit (m-Z)
Compile Buffer (m-X)
Compile Changed Definitions (m-X)
Compile Changed Definitions Of Buffer (m-sh-C, m-X)
Compile File (m-X)
Compile Region (c-sh-C, m-X)
Compiler Warnings (m-X)
Edit Compiler Warnings (m-X)
Evaluate And Exit (c-m-Z)
Evaluate And Replace Into Buffer (m-X)
Evaluate Buffer (m-X)
Evaluate Changed Definitions (m-X)
Evaluate Changed Definitions Of Buffer (m-sh-E, m-X)
Evaluate Into Buffer (m-X)
Evaluate Minibuffer (ESCAPE)
Evaluate Region (c-sh-E, m-X)
Evaluate Region Hack (m-X)
Evaluate Region Verbose (c-m-sh-E)
Load Compiler Warnings (m-X)
Macro Expand Expression (c-sh-M, m-X)
Trace (m-X)
Quit (c-Z)

8.4.6 Lisp Facilities for Finding Out About Lisp

(apropos *string* _package_ _inferiors_ _superiors_**)**
(arglist *function* _flag_**)**
(describe *object***)**
(describe-area *area-name***)**
(describe-defstruct *instance* _structure-name_**)**
(describe-flavor *flavor-name***)**
(describe-package *package-name***)**
(describe-system *system-name***)**
(disassemble *function***)**
(documentation *function***)**
(si:flavor-allowed-init-keywords *flavor-name***)**
(inspect *object***)**
(compiler:load-compiler-warnings *file* _flush-flag_**)**
(mexp)
(trace _specs_**)**
(untrace _specs_**)**
(variable-boundp *variable***)**
(what-files-call *string* _package_**)**
(where-is *symbol* _package_**)**
(who-calls *symbol* _package_ _inferiors_ _superiors_**)**

8.5 Reference Description of Help Functions

This section contains a summary paragraph of documentation for each of the information-finding commands and functions appearing in the summary lists of this document.

This reference list is in alphabetical order by name of the command or function. Zmacs editor commands appear according to the names of the commands that implement them, rather than according to the names of the keys that invoke them. For example, m-X Compile Buffer appears under "C" rather than under "M"; c-sh-A appears under "Q" (because its name is Quick Arglist) rather than under "C". For commands that are usually invoked by a single key rather than by m-X, the key name appears with the command. (Remember you can always use HELP W to find a key name.)

Some Zmacs commands come in pairs, for example, List Callers and Edit Callers. The commands are very similar. The List version allows you to just look at the list or to decide to start editing the items in the list. The list items are always mouse sensitive. For the Edit version of the command, c-. is always the command for moving to the next item.

Apropos (HELP A, m-X)

Displays all the Zmacs commands whose names contain a specified substring. You type the substring. Zmacs displays one line of documentation for the command and which key invokes it in the current context, if any.

(apropos *string* package inferiors superiors**)**

Displays all of the symbols whose print names contain the string. By default, it looks in the global package and its descendants, but you can specify a package name. For symbols that have function bindings, it displays the argument list. For symbols that are bound, it displays a notation "Bound". **apropos** returns the list of symbols that it found.

```
(apropos "forward" 'zwei)
```

(arglist *function* *flag***)** (see also Quick Arglist)

Returns a representation of the arguments that the function expects. When the original function definition contained an **arglist** declaration, **arglist** returns that list when *flag* is not specified or **nil**. When *flag* is not **nil**, then **arglist** returns the real argument list from the function. When the original function used a **values** declaration, **arglist** returns the names for the values returned by the function.

```
(arglist 'make-array)
```

You cannot use **arglist** to find the arguments for combined methods.

Break (SUSPEND) Enters a Lisp Listener from the current window. It uses the screen area of the frame that was selected when you used SUSPEND. When you use it from the editor, any Lisp forms you type are evaluated in the current package (the one showing in the status line). Use RESUME to return to the original context.

c-m-sh-E See Evaluate Region Verbose.

c-sh-A See Quick Arglist.

c-sh-C See Compile Region.

c-sh-D See Long Documentation.

c-sh-E See Evaluate Region.

c-sh-V See Describe Variable At Point.

Compile And Exit (m-Z)

Compiles the buffer and returns from top level. It selects the window from which the last **(ed)** function or the last debugger c-E command was executed.

Compile Buffer (m-X)

Compiles the entire buffer. With a numeric argument, it compiles from point to the end of the buffer. (This is useful for resuming compilation after a prior Compile Buffer has failed.)

Compile Changed Definitions (m-X)
Compiles any definitions that have changed in any Lisp mode buffers. With a numeric argument, it queries individually about whether to compile each changed definition.

Compile Changed Definitions Of Buffer (m-sh-C, m-X)
Compiles any definitions in the current buffer that have been changed. With a numeric argument, it prompts individually about whether to compile each changed definition.

Compile File (m-X) Compiles a file, offering to save it first. It prompts for a file name in the minibuffer, using the file associated with the current buffer as the default. It offers to save the file if the buffer has been modified.

Compile Region (c-sh-C, m-X)
Compiles the region, or if no region is defined, the current definition.

Compiler Warnings (m-X) (see also Edit Compiler Warnings)
Puts all pending compiler warnings in a buffer and selects that buffer. It loads the compiler warnings database into a buffer called *Compiler-Warnings-1*, creating that buffer if it does not exist.

(describe *object***)** (see also **inspect**)
Displays available information about an object, in a format that depends on the type of the object. For example, describing a symbol displays its value, definition, and properties. **describe** returns the object.

```
(describe 'time:get-time)
```

(describe-area *area-name***)**
Displays attributes of the specified area.

```
(describe-area (%area-number 'foo))
(describe-area 'working-storage-area)
```

(describe-defstruct *instance* *structure-name***)**
Displays a description of the instance, showing the contents of each of its slots. *structure-name* is not necessary for named structures but must be provided for unnamed structures. When you supply **structure-name**, you force the function to use that structure name instead of letting the system figure it out; in addition, it overrides the **:describe** option for structures that know how to describe themselves.

Describe Flavor (m-X) (see also **describe-flavor**)

Displays a description of a flavor. It reads a flavor name via the mouse or from the minibuffer using completion. It displays a description of the flavor in a typeout window. The description includes names of flavors that the specified one directly depends on and names of flavors that depend on it. It also displays the documentation and the names of its instance variables.

(describe-flavor *flavor-name***)** (see also Describe Flavor)
Displays descriptive information about a flavor.

```
(describe-flavor 'tv:basic-menu)
```

(describe-package *package-name***)**
Displays information about a package.

```
(describe-package 'zwei)
```

That example is the same as this one:

```
(describe (pkg-find-package 'zwei))
```

(describe-system *system-name***)**
Displays information about a system, including the name of the file containing the system declaration and when the files in the current version of the system were compiled.

Describe Variable (m-X)
Displays the documentation and current value for a Zmacs variable. It reads the variable name from the minibuffer, using completion.

Describe Variable At Point (c-sh-V)
Displays information, in the echo area, about the current Lisp variable. The information includes whether the variable is declared special, whether it has a value, what file defines it, and whether it has documentation put on by **defvar** or **defconst**. When nothing is available, it checks for lookalike symbols in other packages.

(disassemble *function***)** (see also **mexp**, Macro Expand Expression)
Displays the macro-instructions for the function. It does not work for functions that are not compiled or that are implemented in microcode, like **cons** or **car**.

```
(disassemble 'plus)
```

Use this function for things like finding clues about whether a macro is being expanded correctly.

Edit Buffers (m-X) (see also List Buffers)
Displays a list of all buffers, allowing you to save or delete buffers and to select a new buffer. A set of single character subcommands lets you specify various operations for the buffers. For example, you can mark buffers to be deleted, saved, or not

modified. Use HELP to see further explanation. The buffer is read-only; you can move around in it by searching and with commands like c-N or c-P.

Edit Callers (m-X) (see also List Callers, Multiple Edit Callers)
Prepares for editing all functions that call the specified one. It reads a function name via the mouse or from the minibuffer with completion. By default, it searches the current package. You can control the package being searched by giving the function an argument. With c-U, it searches all packages; with c-U c-U, it prompts for a package name. It selects the first caller; use c-. (Next Possibility) to move to a subsequent definition. It takes about 5 minutes to search all packages.

Edit Changed Definitions (m-X) (see also List Changed Definitions)
Determines which definitions in any Lisp mode buffer have changed and selects the first one. It makes an internal list of all the definitions that have changed in the current session and selects the first one on the list. Use c-. (Next Possibility) to move to a subsequent definition. Use a numeric argument to control the starting point for determining what has changed:

1	For each buffer, since the file was last read (the default).
2	For each buffer, since it was last saved.
3	For each definition in each buffer, since the definition was last compiled.

Edit Changed Definitions Of Buffer (m-X) (see also List Changed Definitions Of Buffer)
Determines which definitions in the buffer have changed and selects the first one. It makes an internal list of all the definitions that have changed since the buffer was read in and selects the first one on the list. Use c-. (Next Possibility) to move to subsequent definitions. Use a numeric argument to control the starting point for determining what has changed:

1	Since the file was last read (the default).
2	Since the buffer was last saved.
3	Since the definition was last compiled, for each definition in the buffer.

Edit Combined Methods (m-X) (see also List Combined Methods)
Prepares to edit the methods for a specified message to a specified flavor. It prompts first for a message name, then for a flavor name. It selects the first combined method component. Use c-. (Next Possibility) to move to a subsequent definition. The definitions appear in the order that they would be called when the message was sent. Error messages appear when the flavor does not handle the message and when the flavor requested is not a composed, instantiated flavor.

Edit Compiler Warnings (m-X) (see also Compiler Warnings)

Prepares to edit all functions whose compilation caused a warning message. It queries, for each of the files mentioned in the database, whether you want to edit the warnings for the functions in that file. It splits the screen, putting the warning message in the top window. The bottom window displays the source code whose compilation caused the message. Use c-. (Next Possibility) to move to a subsequent warning and source function. After the last warning, it returns the screen to its previous configuration.

Edit Definition (m-.)

Prepares to edit the definition of a function, variable, flavor, or anything else defined with a "defsomething" special form. It prompts for a definition name from the minibuffer. Name completion is available for definitions in files that have already been loaded into buffers. You can select a name by clicking the mouse over a definition name in the current buffer. It selects the buffer containing the definition for that name, first reading in the file if necessary. With a numeric argument, it selects the next definition that satisfies the most recent name given. It tells you in the echo area when it finds more than one definition for a name.

Edit File Warnings (m-X)

Prepares to edit any functions in a specified file for which warnings exist. It prompts for a file name, which can be either a source file or a compiled file. It splits the screen, putting a warning message from the warnings database in the top window. The bottom window displays the source code whose compilation caused the message. If the database does not contain any warnings for this file, it prompts for the name of a file containing the warnings. Use c-. (Next Possibility) to move to a subsequent warning and source function. After the last warning, it returns the screen to its previous configuration.

Edit Methods (m-X) (see also List Methods)

Prepares to edit all the methods on any flavor for a particular message. It prompts for a message name. It finds all the flavors with handlers for the message, makes an internal list of the method names, and selects the definition for the first one. Use c-. (Next Possibility) to move to subsequent definitions.

Edit Zmacs Command (m-X)

Finds the source for the function installed on a key. You can press any key combination or enter an extended command name. Use a numeric argument to edit the function that implements a prefix command (like m-X or c-X).

Evaluate And Exit (c-m-Z)
Evaluates the buffer and returns from top level. It selects the window from which the last **ed** function or the last debugger c-E command was executed.

Evaluate And Replace Into Buffer (m-X)
Evaluates the Lisp object following point in the buffer and replaces it with its result.

Evaluate Buffer (m-X)
Evaluates the entire buffer. With a numeric argument, it evaluates from point to the end of the buffer.

Evaluate Changed Definitions (m-X)
Evaluates any definitions that have changed in any buffers. With a numeric argument, it prompts individually about whether to evaluate particular changed definitions.

Evaluate Changed Definitions Of Buffer (m-sh-E, m-X)
Evaluates any definitions in the current buffer that have been changed. With a numeric argument, it prompts individually about whether to evaluate particular changed definitions.

Evaluate Into Buffer (m-X)
Evaluates a form read from the minibuffer and inserts the result into the buffer. You enter a Lisp form in the minibuffer, which is evaluated when you press END. The result of evaluating the form appears in the buffer before point. With a numeric argument, it also inserts any typeout that occurs during the evaluation into the buffer.

Evaluate Minibuffer (m-ESCAPE)
Evaluates forms from the minibuffer. You enter Lisp forms in the minibuffer, which are evaluated when you press END. The value of the form itself appears in the echo area. If the form displays any output, that appears as a typeout window.

Evaluate Region (c-sh-E, m-X)
Evaluates the region. When no region has been defined, it evaluates the current definition. It shows the results in the echo area.

Evaluate Region Hack (m-X)
Evaluates the region, ensuring that any variables appearing in a **defvar** have their values set. When no region has been defined, it evaluates the current definition. It shows the results in the echo area.

Evaluate Region Verbose (c-m-sh-E)
Evaluates the region. When no region has been defined, it evaluates the current definition. It shows the results in a typeout window.

(flavor-allowed-init-keywords *flavor-name***)** (In **si:**)
Returns a list containing the init keywords and inittable instance variables allowed for a particular flavor.

```
(si:flavor-allowed-init-keywords 'tv:basic-menu)
```

Function Apropos (m-X)
Displays all the Lisp functions whose print names contain a particular substring. It reads the substring from the minibuffer. By default, it searches the current package. You can control the package being searched by giving the function an argument. With c-U, it searches all packages; with c-U c-U, it prompts for a package name.

(inspect *object***)** (see also **describe**)
Creates or selects an Inspector window and displays available information about an object. **inspect** and **describe** provide similar information, except that **inspect** is an interactive facility for further exploring a data structure.

```
(inspect tv:selected-window)
(inspect (tv:window-under-mouse))
```

List Buffers (c-X c-B) (see also Edit Buffers)
Prints a list of all the buffers and their associated files. The lines in the list are mouse sensitive, offering a menu of operations on the buffers. Clicking left on a line selects the buffer. For buffers with associated files, the version number of the file appears. For buffers without associated files, the size of the buffer in lines appears. For Dired buffers, the pathname used for creating the buffer appears as the version. The list of buffers appears sorted in order of last access, with the currently selected one at the top. You can inhibit sorting by setting **zwei:*sort-zmacs-buffer-list*** to **nil**.

List Callers (m-X) (see also Edit Callers, Multiple List Callers)
Lists all functions that call the specified function. It reads a function name via the mouse or from the minibuffer with completion. By default, it searches the current package. You can control the package being searched by giving the function an argument. With c-U, it searches all packages; with c-U c-U, it prompts for a package name. The names are mouse sensitive. Use c-. (Next Possibility) to start editing the definitions in the list. It takes about 5 minutes to search all packages.

List Changed Definitions (m-X) (see also Edit Changed Definitions)
Displays a list of any definitions that have been edited in any buffer. Use c-. (Next Possibility) to start editing the definitions in the list. Use a numeric argument to control the starting point for determining what has changed:

1 For each buffer, since the file was last read (the default).
2 For each buffer, since it was last saved.
3 For each definition in each buffer, since the definition was last compiled.

List Changed Definitions Of Buffer (m-X) (see also Edit Changed Definitions Of Buffer)
Displays the names of definitions in the buffer that have changed. It makes an internal list of the definitions changed since the buffer was read in and offers to let you edit them. Use c-. (Next Possibility) to move to subsequent definitions. Use a numeric argument to control the starting point for determining what has changed:
1 Since the file was last read (the default).
2 Since the buffer was last saved.
3 Since the definition was last compiled, for each definition in the buffer.

List Combined Methods (m-X) (see also Edit Combined Methods)
Lists the methods for a specified message to a specified flavor. It prompts first for a message name, then for a flavor name. It lists the names in a typeout window. Error messages appear when the flavor does not handle the message and when the flavor requested is not a composed, instantiated flavor. Use c-. (Next Possibility) to start editing the definitions in the list.

List Commands (m-X)
Lists names and one-line summaries for all extended commands available in the current context. It displays the names in a typeout window. The list is not sorted.

List Definitions (m-X)
Displays the definitions from a specified buffer. It reads the buffer name from the minibuffer, using the current buffer as the default. It displays the list as a typeout window. The individual definition names are mouse sensitive.

List Matching Lines (m-X)
Displays all the lines following point in the current buffer that contain a given string. It prompts for the string in the minibuffer. With a numeric argument, it shows only the first *n* occurrences of the string following point. The lines are mouse sensitive.

List Matching Symbols (m-X)
Lists the symbols that satisfy a predicate. It prompts for a predicate lambda expression of one argument. The predicate gets compiled, for speed. The predicate must return something other than **nil** for the symbol to be included in the list. For example

you pressed: `m-X L M S`
minibuffer contains: `'(LAMBDA (SYMBOL))`
revised minibuffer: `'(LAMBDA (SYMBOL) (string-search "foo" symbol))`

By default, it searches the current package. You can control the package being searched by giving the function an argument. With c-U, it searches all packages; with c-U c-U, it prompts for a package name. It selects the first one; use c-. (Next Possibility) to move to a subsequent definition.

List Methods (m-X) (see also Edit Methods)
Lists all the function specs for all methods on any flavor that handle a particular message. It prompts for the message name. It finds all the flavors with methods for the message and displays the information in a typeout window. The function specs are mouse sensitive.

List Registers (m-X)
Displays names and contents of all defined registers. Use Apropos to see commands for manipulating registers.

List Some Word Abbrevs (m-X)
Lists the abbreviations or expansions that contain the given string. Use Apropos to see the other abbreviation commands.

List Tag Tables (m-X)
Lists the names of all the tag tables currently available. Use Apropos to see other commands using tags.

List Variables (m-X)
Lists all Zmacs variable names and their values. With a numeric argument, it also displays the documentation line for the variable. Zmacs variables are those that have been provided for customizing the editor. Their names differ from their internal Lisp names:

Zmacs variable name: Fill Column
Internal Lisp name: **zwei:*fill-column***

List Word Abbrevs (m-X)
Lists all current abbreviations and their expansions.

(load-compiler-warnings *file* *<u>flush-flag</u>***)** (In **compiler:**) (see also Load Compiler Warnings)
Loads a file containing compiler warning messages into the warnings database. It expects to load a file containing the printed representation of compiler warnings (as saved by **print-compiler-warnings**). It uses *flush-flag* to determine whether to replace any of the warnings already in the database. When the flag is not **nil**, it deletes any warnings associated with a source file before loading any new warnings for that file. Otherwise, it merges warnings from the file with those already in the warnings database. The default is **t**.

Load Compiler Warnings (m-X) (see also **compiler:load-compiler-warnings**)
Loads a file containing compiler warning messages into the warnings database. It prompts for the name of a file that contains the printed representation of compiler warnings. It always replaces any warnings already in the database.

Long Documentation (c-sh-D) (see also Show Documentation)
Displays the summary documentation for the specified Lisp function. It prompts for a function name, which you can either type in or select with the mouse. The default is the current function.

m-.	See Edit Definition.
m-ESCAPE	See Evaluate MiniBuffer.
m-sh-A	See Show Documentation Function.
m-sh-C	See Compile Changed Definitions Of Buffer.
m-sh-D	See Show Documentation.
m-sh-E	See Evaluate Changed Definitions Of Buffer.
m-sh-F	See Show Documentation Flavor.
m-sh-V	See Show Documentation Variable.

Macro Expand Expression (c-sh-M, m-X)
Displays the macro expansion of the next Lisp expression in the buffer. It reads the Lisp expression following point and expands all macros within it at all levels, displaying the result on the typeout window. With a numeric argument, it pretty-prints the result back into the buffer, immediately following the expression.

(mexp) (see also **disassemble**)
Displays the expansion of a macro. It prompts for a macro invocation to expand and then displays its expansion without evaluating it. It continues prompting until you enter something that is not a cons (for example, () stops it.)

Multiple Edit Callers (m-X) (see also Edit Callers)
Prepares for editing all functions that call the specified ones. It reads a function name from the minibuffer, with completion. It then keeps asking for another function name until you end it with just RETURN. By default, it searches the current package. You can control the package being searched by giving the function an argument. With c-U, it searches all packages; with c-U c-U, it prompts for a package name. It selects the first caller; use c-. (Next Possibility) to move to a subsequent definition.

Multiple List Callers (m-X) (see also List Callers)
Lists all the functions that call the specified functions. It reads a

function name from the minibuffer, with completion. It continues prompting for a function name until you end it with just RETURN. By default, it searches the current package. You can control the package being searched by giving the function an argument. With c-U, it searches all packages; with c-U c-U, it prompts for a package name. Use c-. (Next Possibility) to start editing the definitions in the list.

Print Modifications (m-X)
: Displays the lines in the current buffer that have changed since the file was first read into a buffer. With a numeric argument, it displays the lines that have changed since the last save. To provide context, it shows the first line of each section that contains a change, whether or not that line has changed. The lines are mouse sensitive, allowing you to move to the location of a change.

Quick Arglist (c-sh-A) (see also **arglist**)
: Displays the argument list for the current function. With a numeric argument, it reads the function name via the mouse or from the minibuffer. When the original function uses a **values** declaration, Quick Arglist returns the names for the values returned by the function.

Quit (c-Z)
: Returns from top level. It selects the window from which the last **(ed)** function or the last debugger c-E command was executed.

Select Some Buffers as Tag Table (m-X)
: Creates a tag table by selecting some buffers currently read in, querying about each one. With a numeric argument, it asks only about buffers whose name contains a given string.

Select System as Tag Table (m-X)
: Creates a tag table for all the files in a system. It uses the file names as they appear in the **defsystem** function for that system.

Show Documentation (m-X, m-sh-D)
: Looks up a topic from the documentation database and displays it on a typeout window. It offers the current definition as a default, but prompts for a definition, which can be supplied by mouse or minibuffer. It accepts only those topics for which documentation has been installed.

Show Documentation Flavor (m-sh-F)
: Displays the documentation for the current flavor. With a numeric argument, it prompts for a device. The devices currently supported are the screen and an LGP printer.

Show Documentation Function (m-sh-A)
: Displays the documentation for the current function. With a

numeric argument, it prompts for a device. The devices currently supported are the screen and an LGP printer.

Show Documentation Variable (m-sh-V)
Displays the documentation for the current variable. With a numeric argument, it prompts for a device. The devices currently supported are the screen and an LGP printer.

Tags Search (m-X) Searches all files in a tags table for a specified string. It reads the string from the minibuffer and then prompts for a tags table name.

Trace (m-X) (see also **untrace**)
Toggles tracing for a function. With a numeric argument, it simply enables tracing for some function, without prompting you for trace options. It uses the same interface for specifying options as the Trace program in the System menu. See the section "Tracing Function Execution" in *Program Development Utilities*.

(trace specs**)** (see also **untrace**)
Turns on tracing for a function. With no arguments, it returns a list of all things currently being traced. With no additional options, tracing displays the name and arguments for a function each time it is called and its name and value(s) each time it returns. Complex options are available for entering breakpoints or executing code conditionally during tracing. See the section "Tracing Function Execution" in *Program Development Utilities*. See the section "Trace" in *Text Editing and Processing*.

```
(trace foo bar)
(trace #'(:method command-found :push))
```

Tracing very common functions (like **format**) or functions used by **trace** itself or by the scheduler (like **time:get-time**) can crash the machine.

(untrace specs**)** Turns off tracing for a function that is being traced. With no argument, it turns off tracing for all functions currently being traced.

(variable-boundp *variable***)**
Returns **nil** or **t** indicating whether or not the variable is bound.

```
(variable-boundp tv:current-window)
```

(what-files-call *symbol* package**)**
Displays the names of files that contain uses of *symbol* as a function, variable, or constant. It searches all the function cells of all the symbols in *package*. By default, it searches the global package and its descendants. It returns a list of the pathnames of the files containing the callers.

Where Is Symbol (m-X)
: Displays the names of packages that contain symbols with the specified name.

(where-is *string* *<u>package</u>***)**
: Displays the names of all packages that contain a symbol whose print name is *string*. It ignores the case of string. By default, it looks in the global package and its descendants. **where-is** returns a list of the symbols that it finds.

```
(where-is "foobar")
```

(who-calls *symbol* *<u>package</u>* *<u>inferiors</u>* *<u>superiors</u>***)**
: Displays a line of information about uses of the symbol as a function, variable, or constant. It searches all the function cells of all the symbols in *package*. By default, it searches the global package and its descendants. It returns a list of the names of the callers.

```
(who-calls 'time:get-time 'hacks)
```

8.6 Editing Your Input

When you make a mistake in typing or change your mind when typing a command or expression to the system, you have two choices:

- Press ABORT and begin again.
- Edit your input.

You do not need to invoke the input editor explicitly. RUBOUT and many simple Zmacs commands such as c-A, c-D, c-E, m-F, and m-B are always available.

A history of all the commands you have typed is maintained and you can recall previous commands for editing and re-submission. See the section "Command History", page 15.

8.6.1 How the Input Editor Works

The input editor is a feature of all interactive streams, that is, streams that connect to terminals. Its purpose is to let you edit minor mistakes in typein. At the same time, it is not supposed to get in the way; Lisp is to see the input as soon as you have typed a syntactically complete form. The definition of "syntactically complete form" depends on the function that is reading from the stream; for **read**, it is a Lisp expression. This section describes the general protocol used for communication between the input editor and reading functions such as **read** and **readline**.

By *reading function* we mean a function that reads a number of characters from a stream and translates them into an object. For example, **read** reads a Lisp expression and returns an object. **readline** reads a line of characters and returns a string as its first value. Reading functions do not include the more primitive **:tyi** and **:any-tyi** stream operations, which take and return one character or blip from the stream.

The tricky thing about the input editor is the need for it to figure out when you are all done. The idea of an input editor is that as you type in characters, the input editor saves them up in an *input buffer* so that if you change your mind, you can edit them and replace them with different characters. However, at some point the input editor has to decide that the time has come to stop putting characters into the input buffer and let the reading function start processing the characters. This is called "activating".

The right time to activate depends on the function calling the input editor, and determining it may be very complicated. If the function is **read**, figuring out when one Lisp expression has been typed requires knowledge of all the various printed representations, what all currently defined reader macros do, and so on. The input editor should not have to know how to parse the characters in the input buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The input editor interface is organized so that the calling function can do all the parsing, while the input editor does all the handling of editing commands, and the two are kept completely separate.

Following is a summary of how the input editor works. The input editor used to be called the rubout handler, and some operations and variables still have "rubout-handler" in their names.

When a reading function is called to read from a stream that supports the **:input-editor** operation, that function "enters" the input editor. It then goes ahead **:tyi**'ing characters from the stream. Because control is inside the input editor, the stream echoes these characters so the user can see the input. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the input editor it is also handing it the responsibility for echoing). The input editor is also saving all these characters in the input buffer, for reasons disclosed in the following paragraph. When the reading function decides it has enough input, it returns and control "leaves" the input editor. That was the easy case.

If you press RUBOUT or a keystroke that represents another editing command, the input editor processes the command and lets you insert characters before the last one in the line. The input editor modifies the input buffer and the screen accordingly. Then, when you type the next nonediting character at the end of the line, a **throw** is done, out of all recursive levels of **read**, reader macros, and so forth, back to the point where the input editor was entered. Now the **read** is tried over again, rereading all the characters you had typed and not rubbed out, but not echoing them this time. When the saved characters have been exhausted, additional input is read from you in the usual fashion.

The input editor has options that can cause the **throw** to occur at other times as well. With the **:activation** option, when you type an activation character a **throw** occurs, a rescan is done if necessary, and a final blip is returned to the reading function. With the **:preemptable** and **:command** options, a blip or special character in the input stream causes control to be returned from the input editor immediately, without a rescan. These options let you process mouse clicks or special keystroke commands as soon as they are read.

The effect of all this is a complete separation of the functions of input editing and parsing, while at the same time mingling the execution of these two functions in such a way that input is always "activated" at just the right time. It does mean that the parsing function (in the usual case, **read** and all macro-character definitions) must be prepared to be thrown through at any time and should not have nontrivial side-effects, since it may be called multiple times.

If an error occurs while inside the input editor, the error message is printed and then additional characters are read. When you press RUBOUT, it rubs out the error message as well as the last character. You can then proceed to type the corrected expression; the input is reparsed from the beginning in the usual fashion.

9. When and How to Use the Garbage Collector

The garbage collector is a program in the Symbolics-Lisp system that automatically finds, tracks, and recovers memory occupied by unused objects (garbage) in the current Lisp world. It is a particular implementation of *automatic storage management*, meaning that programmers (and also nonprogrammer users of the system) can do things that allocate, use, and discard large amounts of virtual memory, without having to pay any attention to the management of the memory. In systems without this feature, most large-scale uses of virtual memory have to be managed "manually" (under control of a user program); manual storage management is difficult and error-prone because it is quite difficult for a program to "prove" that an object really is of no use to any other system component.

Automatic storage management also has the desirable effect of lengthening the "session" you spend with a particular world between cold boots. Without it, most normal uses of a Lisp system will exhaust virtual memory rather quickly. With it, normal use (whether or not for programming) is longer and more convenient.

When the usual, incremental garbage collector is operating, the Scavenger periodically goes through virtual memory, looking for objects that can be proven not to be garbage. These "good" objects are transported to a safe place, and the memory occupied by the garbage is reclaimed automatically. In the meantime, new objects can still be created. (More extensive information on automatic storage management is available elsewhere; See the section "Operation of the Garbage Collector" in *Internals, Processes, and Storage Management.*)

There are different kinds of garbage collection available in Symbolics-Lisp. All require some additional virtual memory for their own use. Until the scavenging process is complete, running with the garbage collector can require up to twice as much space as running without the garbage collector (depending on how much of old space was garbage, compared to how much had to be copied). If you have been running without the garbage collector for a long time, you might not have enough room to successfully run the garbage collector and collect all the garbage. If the garbage collector is not operating, the system sends notifications as you approach a certain percentage full. See the section "Storage Requirements for Garbage Collection" in *Internals, Processes, and Storage Management.*

The command Show GC Status allows you to check on how much free space you have and determine whether or not you should turn on the garbage collector.

```
Show GC Status

Status of the ephemeral garbage collector:                On
First level of WORKING-STORAGE-AREA: capacity 196K, 416K allocated, 10K used.
Second level of WORKING-STORAGE-AREA: capacity 98K, 256K allocated, 137K used.
```

```
Status of the dynamic garbage collector:                    On
Dynamic (new+copy) space 1,746,767.  Old space 0.  Static space 6,856,801.
Free space 6,957,056.  Committed guess 6,559,133, leaving 135,779 to use
  before flipping.
There are 2,343,001 words available before (GC-IMMEDIATELY) might run out of
space.
Doing (GC-IMMEDIATELY) now would take roughly 14 minutes.
There are 6,957,056 words available if you elect not to garbage collect.

Garbage collector process state: Await ephemeral or dynamic full
Scavenging during cons: On, Scavenging when machine idle: On
The GC generation count is 2 (1 full GC, 0 dynamic GC's, and 1 ephemeral GC).
Evaluate (CHOOSE-GC-PARAMETERS) to examine or modify the GC parameters.
```

The command Start GC turns on the garbage collector.

Start GC *keywords*

Turns on the garbage collector.

keywords can be:

:dynamic	Dynamic Level of incremental GC.
:ephemeral	Ephemeral Level of incremental GC.
:immediately	Perform a complete garbage collection right now.

Start GC :ephemeral is recommended for general purposes. This cleans up after you as you work, keeping virtual memory requirements for garbage collecting to a minimum. When, in spite of scavenging, enough garbage has accumulated, you receive a notification. At that point you can use Start GC :immediately to do a complete garbage collection. See the section "Ephemeral-object Garbage Collection" in *Internals, Processes, and Storage Management.*

10. How to Get Hardcopy

10.1 Commands for Producing Hardcopy

You can produce hardcopy using the System Menu, from the editor, from Zmail, from Dired in the editor, and from the file system editor. You can also get a hardcopy of your screen at any time.

In order for menu items, commands, and functions that refer to printing and hardcopy to work, your site must have a properly connected printing device. Printers are objects in the namespace database. See the section "Namespace System Printer Objects" in *Networks.*

10.1.1 Hardcopying From the System Menu

To produce hardcopy using the System Menu, click on [Hardcopy]. This pops up a menu that allows you to specify the pathname of the file to be hardcopied, the printer to send it to, and some options for format.

10.1.2 Hardcopying From Zmacs

You can hardcopy a region, a buffer, or a file from Zmacs.

Hardcopy Region (m-X)

Sends a region's contents to the local hardcopy device for printing.

Hardcopy Buffer (m-X)

Prompts for the name of a buffer and then prints the specified buffer on the local hardcopy device.

Hardcopy File (m-X)

Sends a file to the local hardcopy device for printing.

Kill Or Save Buffers (m-X)

Puts up a multiple-choice menu listing all existing buffers. Choices are: Save, Kill, Unmodify, and Hardcopy. Specify these options next to the buffer names in the menu. This command appears on the editor menu.

10.1.3 Hardcopying From Zmail

You can hardcopy a single message or a collection of messages from Zmail.

Action	*Command*
One message	[Move / Hardcopy] Click right on its summary line then [Move / Hardcopy]
All messages in current sequence	[Map over / Move / Hardcopy]

In any of these commands you can use [Hardcopy (R)] to get a menu that permits you to specify the number of copies, the font, and which printer to use. The `Other` option in the list of printers allows you to specify an arbitrary printer, using either its pretty name or its namespace name. This printer becomes the selected printer, and remains in the menu for subsequent hardcopy commands.

10.1.4 Hardcopying From Dired

You can mark files to be hardcopied in Dired. When you exit from Dired, the files marked to be hardcopied are sent to the printer.

`P` Dired Hardcopy File

Marks the current file for printing. Dired puts a P in the first column to show that the file has been so marked.

With a numeric argument *n*, marks the next *n* files for printing.

10.1.5 Hardcopying the Screen

You can get a hardcopy of what is displayed on your screen by pressing `FUNCTION Q`:

`Q`	Hardcopies the entire screen.
`c-Q`	Hardcopies the selected window.
`m-Q`	Hardcopies the entire screen, minus the status and mouse documentation lines.

10.1.6 Hardcopying From the File System Editor

You can invoke the system hardcopy menu from FSEdit. You click on `Hardcopy` in the menu of file operations invoked by clicking right on a file name.

10.2 Checking the Status of Hardcopy Devices

You can find out the status of a hardcopy device in Zmacs:

Show Hardcopy Status (m-X)

Show the status of a hardcopy device or all of them.

11. Understanding Networks and the Namespace System

11.1 Introduction to the Namespace System

When computers are connected by means of networks to form a distributed computing environment, the computers should all be able to share information that describes that environment. For example, all the computers need to know or be able to find out about the names and addresses of the other computers to which they can communicate. A personal workstation computer might want to know what printers are available on which server computers. A computer trying to send mail to a particular user might want to know on which computer that user's mailbox resides on.

The Symbolics Lisp Machine system has a convention by which such information can be maintained and shared in a simple database. The database is maintained by a *namespace server*. Other systems can query or make changes to the database by communicating over the network with the server. This database is the *namespace database*, a specific example of a distributed network database. Both the more specific term *namespace database* and the generic term *network database* are used to refer to it. However, in general, *namespace database* refers to the Symbolics implementation of network databases and *namespace system* refers to the namespace database and the tools to use it.

The database is structured to understand that there can be many different networks in a distributed environment, and so there are network objects to represent different networks. Hosts can be on more than one network, and some hosts that are on two networks can serve as gateways from one network to the other. One of the purposes of the database is to let a user host find a path to a server host, using whichever networks and gateways are necessary.

The database is designed so that it is not specific to the Lisp Machine; in theory, any computer system could be made to use the database, and act as a user or server.

The database consists of a collection of *objects*. Each object has

- A class. See the section "Namespace System Classes", page 112.
- Attributes. See the section "Namespace System Attributes", page 112.
- A name. See the section "Names and Namespaces", page 113.

All objects except namespaces themselves are added to the namespace database by using the namespace editor, which is invoked with Edit Namespace Object (or from Lisp with **tv:edit-namespace-object**). See the section "Updating the Namespace Database", page 116.

11.1.1 Namespace System Classes

Every object has a *class*, which tells what kind of object it is. Every class is identified by a global-name.

The following classes are especially important to the Lisp Machine system:

host	A host object represents any computer, usually connected to a network.
user	A user object represents a person who uses any of the hosts, or a daemon user, for example, a Lisp Machine.
network	A network object represents a computer network, to which some hosts are attached.
printer	A printer object represents a device for producing hardcopy.
site	A site object represents a collection of hosts, printers, and networks that are grouped together in one physical location.
namespace	A namespace object represents a mapping from names of objects to objects.

11.1.2 Namespace System Attributes

Attributes represent characteristics of the object. Each attribute has an *indicator* (the name of the attribute) and a *value;* they work like property lists in Lisp. For example, every host has a system-type (saying which operating system it runs), every printer has a type (saying what type of printer it is), and every user has a personal-name.

Each object class has one or more required attributes. However, most attributes are optional; for example, hosts can optionally have a pretty-name, printers can have a default-font, and a user can have a home-address. Some attributes can occur more than once for a given object; for example, a host object can have multiple addresses if it is attached to more than one network.

Each object has a fixed set of attributes: you cannot create additional attributes.

11.1.3 Data Types of Namespace System Attributes

Each class has attributes that are defined to have specific data types. Since the actual representation of the various types of data represented in the database varies from system to system, the namespace system uses the following system-independent types:

Data type	*Value*
object-class	An object in the database, for example, a site object. See the section "Namespace System Classes", page 112.
name	A name in some namespace; name is not shared by all namespaces.
global-name	A name which is not specific to a particular namespace but is shared by all namespaces.
token	An arbitrary character string.
set	An ordered set of elements of the same data type. For example, a value can be a set of names or a set of triples.
pair	A list of two elements of specific data types; each element can be of a different data type.
triple	A list of three elements; each element can be of a different data type.

Name, global-name, and token require simple values, whereas set, pair, and triples require compound values.

Note: The namespace data types specific to the Lisp Machine are described elsewhere: See the section "Namespace System Lisp Data Types" in *Networks*.

11.1.4 Names and Namespaces

Every object has a *name*, which is a character string. Two objects of different classes can have the same name; for example, there can be a printer named george and a user named george; the two are unrelated. An object is identified by its class and its name; if you want to look up an object in the database and you know its name, you have to say "Find the printer named george" or "Find the user named george", not just "Find george".

When long-distance networks are used to link together different sites, however, the possibility of name conflicts arises; that is, two sites may use the same name in the same class for conflicting purposes. For example, suppose you had a host named orange, and you wanted to connect your site over a long-distance network to some other site that happens to have picked the name orange for one of its own hosts. Neither site is forced to change its host names just because it wants to connect to the other site.

To avoid these naming conflicts, the database can include more than one namespace. A *namespace* is a mapping from names to objects, and names in one namespace are unrelated to names in another namespace. (More strictly, it is a mapping from [name, class] pairs to objects, since an object is identified by its class and its name.) Normally each site has one namespace, and the names of all the objects at that site are in that namespace. An object in some other namespace than your own namespace can be referred to using a *qualified name*, which consists of the name of the namespace, a vertical bar, and the name of the object in that namespace.

For example, suppose both Harvard and Yale had computer centers. Harvard has three hosts named yellow, orange, and blue, and Yale has three hosts named apple, orange, and banana. Each computer center would have its own namespace, one named harvard and one named yale. At Harvard, the Harvard computers would be referred to by their unqualified names (yellow, orange, and blue), whereas the Yale computers would be referred to (by users at Harvard) by qualified names (yale|apple, yale|orange, and yale|banana). At Yale it would all work the other way around.

Each namespace also has a list of namespaces called *search rules*. When a name is looked up, each of the namespaces in the search rules list is consulted in turn, until an object of that name is found in one of the namespaces. If you have some other namespace in your search list, it is easier to refer to objects in that namespace, because you do not have to use qualified names unless a name conflict exists.

For example, in the scenario above, the search list for the harvard namespace could have the harvard namespace first and the yale namespace second. Then users at Harvard could refer to Yale's computers as apple, yale|orange, and banana. The qualified name is only necessary because of the name conflict.

Actually, only some classes of objects have names that are in namespaces; other classes of objects are *globally* named, which means that the names are universal, and conflicts are not permitted. In particular, classes, namespaces, and sites are globally named; networks, hosts, printers, and users are named within namespaces. There is never a need for multiply-qualified names; the names of namespaces are global and never need to be qualified themselves.

Some namespaces do not correspond to any local site. Most large nationwide or worldwide networks have their own host naming convention. For example, the Department of Defense Arpanet has its own set of host names, and this is considered a namespace. If a local site includes some hosts that are on the Arpanet, it might want to put the Arpanet namespace into its search list, and install gateways on its Arpanet machine so that other machines on the local network can access the Arpanet.

Some objects can also have *nicknames*. In particular, networks and hosts can have nicknames; objects of other classes cannot. A nickname serves as an alternative name by which the object can be referred. Sometimes you give an object a nickname because its full name is too long to type conveniently, like some host whose name you type frequently. However, each object always has one primary name, which is used when the object is printed.

It is possible for an object to be in several namespaces at once. For example, a host which is on both the Arpanet and a local network at some site might be in both the Arpanet namespace and the local namespace. In this case, each namespace maintains its own separate information on the object. The information from each namespace is merged before being presented to the user.

Note: Search lists are not followed recursively. If a user at Harvard looks up a name and Yale's namespace is in Harvard's search list, Yale's search list is *not* relevant.

11.2 Using the Network

If your machine is on a network and configured properly, you access the network with the terminal program.

The terminal program is available on SELECT T. Since it uses the generic network system, it allows access (in the presence of appropriate gateways) via autodialers to dialups, as well as direct Chaosnet and TCP through a gateway.

The prompt is `Connect to host:`. To this you simply type the name of any host. (Naming of hosts, setting up host databases, declaring host addresses and supported login services are covered in the network documentation.) The network system picks the best login service supported by the host and the optimum route to it. Pressing HELP in response to the initial prompt gives you input editor documentation.

Once connected, commands are given by pressing NETWORK and another single character.

The following commands are available:

A	send an ATTN (in Telnet, a new Telnet "Interrupt Process").
D	Disconnect.
L	Log out of remote host, and break the connection.
Q	Disconnect and deselect this window. (Quit)
M	Toggle MORE processing.

More complicated commands are entered with the extended command, NETWORK X. This command uses a choose variable values window.

NETWORK X provides the capability to change the following:

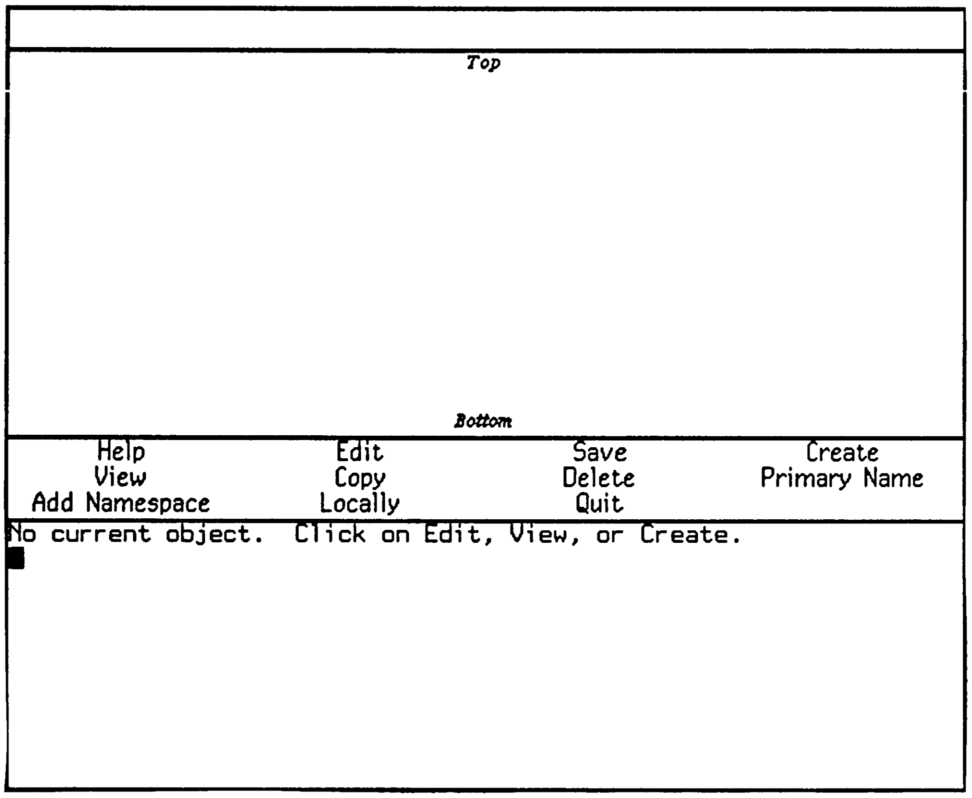
- the escape character
- whether characters overstrike or erase
- whether MORE processing is enabled
- in the case of Telnet, whether Imlac terminal codes are interpreted in host output
- a facility for logging host output to a file (wallpaper).

You must be connected to a host before pressing NETWORK X.

See the section "Using the Terminal Program with Hosts Connected to the Serial Line" in *Reference Guide to Streams, Files, and I/O.*

11.3 Updating the Namespace Database

When you modify the namespace database, you use the command Edit Namespace Object or evaluate the form **(tv:edit-namespace-object)**. Figure 8 shows the initial window.



```
No current object.  Click on Edit, View, or Create.
```

Figure 8. Namespace Object Editor Window

The top pane is where the information about an object is displayed. The bottom pane is for prompting for new information. The middle pane is the command menu. The available commands are:

Help	Displays a brief explanation.

View	Displays information about an object for inspection but not editing.
Edit	Displays information about an object for editing.
Locally	Toggles whether to edit the local or global copy of the information for an object. The initial state is global.
Save	Saves the current information about an object.
Delete	Remove an object from the database.
Create	Add a new object to the database.
Quit	Exit from the namespace editor, without saving the current information. ([Save] should be used before [Quit] when you want to save information.
Copy	Create a new object by copying the current one.
Add Namespace	Add an existing object to a new namespace.
Primary Name	Change the primary name of the current object.

11.3.1 Editing a Namespace Object

To edit an existing namespace object, click on [Edit]. A menu of object classes pops up. Click on the class of object you want to edit. You are prompted for the name of an object to edit. The information for the object is retrieved and displayed in the top window. The fields are mouse sensitive. Clicking on a field prompts you for information in the bottom window. Clicks have the following meaning:

Left	Replace the information in the field.
Middle	Delete information in the field.
Right	Edit the information in the field.

The window can be scrolled. See the section "Scrolling with the Mouse", page 149.

When you are satisfied with the information, click on [Save] to enter it in the database. Click on [Quit] to exit the namespace editor.

11.3.2 Creating a New Namespace Object

To create a new namespace object, click on [Create]. A menu of object classes pops up. Click on the class of object you want to create. A template for the information is displayed in the top window. The fields are mouse sensitive. Clicking on a field prompts you in the bottom window for the information to put in the field.

You can also create a new object by copying an existing object by clicking on [Copy] and then editing the object as appropriate.

The window can be scrolled. See the section "Scrolling with the Mouse", page 149.

When you are satisfied with the information, click on [Save] to enter it in the database. Click on [Quit] to exit the namespace editor.

12. Using the Online Documentation System

12.1 Introduction to the Document Examiner

The *Document Examiner* is a utility for finding and reading documentation.

- Using the Document Examiner is not unlike using the printed documentation. With the printed documentation, you can open a volume to any topic and read through to the end of that topic. With the Document Examiner, you can also "open" the documentation to any topic and read to the end of that topic.

- When you use the Document Examiner, you do not have to remember how information is arranged; for example, you do not have to remember the section, chapter, or printed book in which a particular function is explained. Each function (and each section, chapter, or other division of printed information — even entire books), is directly accessible.

- In addition to looking up documentation, you can create *private documents* with the Document Examiner by placing *bookmarks* in documentation topics and saving the list of bookmarks for future use.

- The online documentation is kept in a *documentation database.* The documentation database consists of documentation binary files. Loaded into your Lisp world is index information about the documentation database.

- Each documentation topic is stored as a record in the documentation database. Each record contains information on a particular topic and is uniquely identified by a topic name. Records fall into two categories: Object records documenting code objects, such as **login** or **tv:menu**, and concept records documenting abstract ideas that are not tied to code, such as "Introduction to the Document Examiner". Records also have a type designation. Examples of object record types are function, flavor, and variable. Concept records have a type of section.

- `SELECT D`, [Document Examiner] in the System Menu, and the command `Select Activity Document Examiner` select the Document Examiner. Pressing `HELP` in the Document Examiner displays a listing of its commands.

12.2 Looking up Documentation

You can look up documentation in the Document Examiner, in an editor (Zmacs, Zmail, Converse), or at a Lisp Listener using a variety of commands. One command looks up and displays documentation by name. Another set of commands pops up a menu of all documentation topic names that satisfy a query request. Such query requests are carried out by matching an initial substring, substrings, or whole words against documentation topic names or their *keywords.* (A keyword is comparable to a word in an index entry.) Clicking on a topic in one of these menus looks up and displays the documentation for that topic. See the section "Documentation Lookup Commands", page 122.

Another set of commands is available for repositioning text in the Document Examiner. See the section "Repositioning Text in the Document Examiner", page 134.

Your lookup request is always made in terms of a documentation topic name. You are prompted for a *type* (section or function, for instance) only when several topics have the same topic name but different types. For instance, suppose there are two topics whose names are "error"; one documents a flavor and the other a function. Requesting a display of "error" causes a menu of the possible types (flavor or function) to pop up. You choose the type you want displayed.

When you look up documentation, the more general the topic you look up, the larger the amount of documentation you see for it. The most general topic names are the names of the books in the printed documentation set. If you are unsure what level in the documentation you need, use the command Find Table Of Contents giving, it the name of the printed book or section that interests you.

In the Document Examiner, you can look at an *overview* of a given topic. The overview includes the topic(s) and book(s) in which the topic appears. In addition, the overview includes a list of keywords included in the topic.

Using the overview facility is a good way to "look back a few pages" in the documentation. Suppose you are looking at the topic "login" and you wonder what the topics before it have to say. You can look at the overview for the topic "login" and see that this topic is included in the topic "Logging In" and in the book "User's Guide to Symbolics Computers". Now you can look at the topic "Logging In".

You can look at a topic's overview by using one of the following methods:

- Click middle on a mouse-sensitive item in the viewer.
- Click middle on a topic in the list of current candidates or the list of bookmarks.
- Use Show Overview at the command prompt and supply the name of a topic.

- Use [Show (M)] in the command menu and supply the name of a topic.

For more information on the overview facility: See the section "Show Overview", page 127.

In addition, you can find the printed book the topic appears in by using What Document (m-X) in the editor.

When you use one of the *documentation find commands* at a Lisp Listener, an editor, or the Document Examiner, a menu of topic names is displayed. This menu includes all the topic names that fulfill your lookup query. When you click on one of the topic names, the chosen topic is displayed. The find commands are Find Initial Substring Candidates, Find Whole Word Candidates, Find Any Candidates, or Find Table Of Contents.

Recovering From a Stuck Document Examiner

When you look up documentation at a Lisp Listener or an editor, the Document Examiner is updated to include the last topic or menu you looked up. This normally happens within a few seconds. Occasionally, the topic you look up in the editor or Lisp Listener does not show up in a matter of seconds in the Document Examiner. If this occurs, enter Peek. In Peek, type P to see a listing of processes. Notice that a process called "DEX background" is showing. This process appears only if there is a problem. Click on this process and select "Debugger" from the menu. You should see a number of proceed options in the debugger, one of which offers to skip trying to process the current topic and move on to the next pending one. Choose that proceed option. The background process that feeds queued topics or candidates lists to the Document Examiner should then "unplug" and put everything that it has been saving into the Document Examiner, one thing at a time.

Topics Pruned From the Documentation Database

When the Documentation Database is installed at your site, the installation manager has the option of pruning the database. By selecting from a menu of major sections in the database, the system manager specifies which files are deleted from the documentation database file server. In this way, space can be saved on the file server by pruning sections not needed at your site.

Trying to display a topic that has been pruned from the database causes a "dummy" topic to be displayed. Suppose, for example, the files containing the section "Streams" were pruned from the database at your site, then trying to display the topic ":tyi" produces the following display:

> **:tyi** *message*
> Documentation for **:tyi** as a Message is offline.
> It appears in document: *Reference Guide to Streams, Files, and I/O*
> Reload the file SYS: DOC; STR; STR2.SAB.4 to make this topic accessible online.

If you decide you do want to see the topic online, you can load the file that contains the index information for the topic. Use **sage:load-index-info**.

12.3 Documentation Lookup Commands

The Document Examiner, the editor, and the command processor all provide various commands for looking up documentation. Some commands are available in all three contexts, while others are available in only one of the contexts. The following are descriptions of the commands provided, categorized according to the context in which the command is available.

Lookup Commands Available in the Document Examiner, Editor, and Command Processor

Show Documentation (an Overview)

Show Documentation looks up a topic and displays it. You can use the command in the Document Examiner, in an editor, and at a command processor.

- In the Document Examiner, Show Documentation prompts for a topic name, with completion, accepting only those topics for which documentation exists in the database. You can use Show Documentation in the Document Examiner any of the following ways:
    - Type the command at the command pane.
    - Use [Show] in the Document Examiner command pane menu.
    - Click left on a mouse-sensitive item in the viewer or an item in the list of candidates or list of bookmarks.
- In an editor, Show Documentation (m-X, m-sh-D) prompts you for a topic name, with completion, accepting only those topics for which documentation exists in the database. You can direct the display of a documentation topic to the local Symbolics LGP by issuing Show Documentation (m-X) with a numeric argument. This pops up a menu offering to display the documentation on the screen or route it to the local Symbolics LGP.
- At a command processor, Show Documentation prompts you for a topic name, with completion, accepting only those topics for which documentation exists in the database. You must enclose multiword topic names in double quotes to provide the name on the command line. You can default the topic name and Show Documentation then prompts you for the name, using **prompt-and-read**, so you do no supply quotes in that case. When you give

the command the keyword argument `:destination`, the command offers to display the documentation on the screen (the default) or route it to the local Symbolics LGP.

It should be noted that topic names for methods are of the form (:method *flavor-name :method-name*), for example, (:method tv:graphics-mixin :point). To look up documentation for methods, use one of the following strategies:

- Use Show Documentation giving it the topic name of the method in the form (:method *flavor-name :method-name*).

- Use Find Whole Word Candidates giving it the name of the method in the form *:method-name*. Then click on the item whose documentation you want to see.

Lookup Commands Available in the Document Examiner and Editor

Find Any Candidates

Sometimes you want to know if the documentation database contains any topics about a particular subject. You might have a string or strings in mind dealing with that subject. Using the command Find Any Candidates, you can search the database for any topics whose topic names or keywords contain the string or strings as *substring(s)*.

A substring is a string that appears somewhere in another string. A substring can be an initial substring. However, when you use the command Find Any Candidates, the search is for a substring that appears anywhere in another string, not necessarily as the initial substring of some word or words in the string. The string "et" is a substring of the strings "set" and "setq". The string "et" is both a substring and an initial substring of the string "etc". The string "et" is not a substring of the strings "est" and "login".

The following situation shows how you can use this command: You want to know if the database contains any topics about setting values of variables. You guess that any such topics would use the string "set" somewhere in their topic names or keywords. So, you use the command Find Any Candidates to search the database for any topics whose topic names or keywords contain the string "set" as a substring. The search returns a list of over 200 candidates.

You can provide the command with a string of several words, for instance, the string "resource window". Note that when the given string contains any space or hyphen characters, the command breaks the string into *tokens* using the space and hyphen characters as delimiters. For example, given the string "resource window", the command breaks it into two tokens, "resource" and "window".

The command looks at all the topic names and keywords in the database listing any in which all the tokens appear as substrings, in effect performing a logical **and** test on the tokens. Given the string "resource window", the command lists several topics, among them the function **defwindow-resource** and the section "The Top-level Function". Both tokens are substrings in the topic name **defwindow-resource** and in the keywords of "The Top-level Function". The order in which you provide the words does not affect the search for topics.

Again, this search is performed not only on the topic names in the documentation database, but also on the keywords listed for each topic. This means that you often find topic names in which the given string does not appear at all. It does, however, appear among the topic's keywords.

In the Document Examiner, the command lists the topic names it has found in the candidates list. At an editor, the list takes the form of a menu.

In an editor you can direct the display of a documentation topic to the local Symbolics LGP by issuing Find Any Documentation with a numeric argument. This pops up a menu offering to display the documentation on the screen or route it to the local Symbolics LGP.

This command is also available as [Find (R)] in the Document Examiner command pane menu.

Find Initial Substring Candidates

Sometimes you want to know if the documentation database contains any topics about a particular subject. You might have an initial substring or substrings in mind dealing with that subject. Using the command Find Initial Substring Candidates, you can search the database for any topics whose topic names or keywords contain the substring or substrings as *initial substring(s).*

An initial substring is a string that appears as the beginning of some string. For example, the string "set" is an initial substring of the string "setq". The string "set" is not an initial substring of the string "reset". The string "set" is a substring of the string "reset".

The following situation shows how you can use this command: You want to know if the database contains any topics about setting values of variables. You guess that any such topics would use the string "set" as an initial substring somewhere in their topic names or keywords. So, you use the command Find Initial Substring Candidates to search the database for any topics whose topic names or keywords contain the string "set" as an initial substring. What the search returns is a list of over 100 candidates.

You can provide the command with a string of more than one word, for instance the string "set-globally". Note that when the given string contains any space or hyphen characters, the command breaks the string into *tokens* using the space and hyphen characters as delimiters. For example, given the string "set globally", the command breaks it into two tokens, "set" and "globally".

The command looks at all the topic names and keywords in the database, listing any in which all the tokens appear as initial substrings (in effect performing a logical **and** test on the tokens). Given the string "set globally", the command lists two topic names, the functions **set-globally** and **setq-globally**. Both tokens are initial substrings in each topic name. The order in which you provide the words does not affect the search for topics.

Again, this search is performed not only on the topic names in the documentation database, but also on the keywords listed for each topic. This means that you often find topic names in which the given string does not appear at all. It does, however, appear among the topic's keywords.

In the Document Examiner, the command lists the topic names it has found in the candidates list. At an editor, the list takes the form of a menu.

Find Initial Substring Candidates treats leading punctuation as part of the word. Thus, asking for initial substring of "area" does not return "*area" or "%area". If you want "anything containing area" you have to use the most general matching command, Find Any Candidates.

In an editor you can direct the display of a documentation topic to the local Symbolics LGP by issuing Find Initial Substring Candidates with a numeric argument. This pops up a menu offering to display the documentation on the screen or route it to the local Symbolics LGP.

This command is also available as [Find (M)] in the Document Examiner command pane menu.

Find Table of Contents

Displays a menu containing the given topic's table of contents. For example, the table of contents of "The Document Examiner" displays as:

```
The Document Examiner
  Introduction to the Document Examiner
  Looking Up Documentation
    Recovering From a Stuck Document Examiner
    Topics Pruned From the Documentation Database
  Documentation Lookup Commands
     Lookup Commands Available in the Document Examiner, Editor, and Command Processor
       Show Documentation (an Overview)
     Lookup Commands Available in the Document Examiner and Editor
       Find Any Candidates
   .
   .
   .
```

You can ask to see a table of contents for any topic; it is not limited to top-level books. The table of contents of "Documentation Lookup Commands" displays as:

```
Documentation Lookup Commands
  Lookup Commands Available in the Document Examiner, Editor, and Command Processor
    Show Documentation (an Overview)
  Lookup Commands Available in the Document Examiner and Editor
    Find Any Candidates
  .
  .
  .
```

This command is also available as [Show (R)] in the Document Examiner command pane menu.

Find Whole Word Candidates

Sometimes you want to know if the documentation database contains any topics about a particular subject. You might have a word or words in mind dealing with that subject. Using the command Find Whole Word Candidates, you can search the database for any topics whose topic names or keywords contain the word or words as *whole* word(s).

A whole word is a string separated from other strings by space or hyphen characters. The string "set" appears as a whole word in the topic name "Creating a Set of Condition Flavors" and in the topic name "set-globally". It does not appear as a whole word in the topic name "setq". In the topic name "setq", the string "set" appears as an initial substring.

The following situation shows how you can use this command: You want to know if the database contains any topics about setting values of variables. You guess that any such topics would use the string "set" in their topic names or keywords. So, you use the command Find Whole Word Candidates to search the database for any topics whose topic names or keywords contain the string "set" as a whole word. The search returns a list of over 70 candidates.

You can provide the command with more than one word. For example, you give the command the string "set globally". Note that when the given string contains any space or hyphen characters, the command breaks the string into *tokens* using the space and hyphen characters as delimiters. For example, given the string "set globally", the command breaks it into two tokens, "set" and "globally".

The command looks at all the topic names and keywords in the database listing any in which all the tokens appear as whole words (in effect performing a logical **and** test on the tokens). Given the string "set globally", the command lists exactly one topic, the function **set-globally**. The order in which you provide the words does not affect the search for topics.

Again, this search is performed not only on the topic names in the documentation database, but also on the keywords listed for each topic. This means that you often find topic names in which the given string does not appear at all. It does, however, appear among the topic's keywords.

In the Document Examiner, the command lists the topic names it has found in the candidates list. At an editor, the list takes the form of a menu.

Find Whole Word Candidates treats leading punctuation as part of the word. Thus, asking for whole word match of "area" does not return "*area" or "%area". If you want "anything containing area" you have to use the most general matching command, Find Any Candidates.

In an editor you can direct the display of a documentation topic to the local Symbolics LGP by issuing Find Whole Word Candidates with a numeric argument. This pops up a menu offering to display the documentation on the screen or route it to the local Symbolics LGP.

This command is also available as [Find] in the Document Examiner command pane menu.

Lookup Commands Available in the Document Examiner

Select Candidate List

Selects from the history of candidates lists, popping up a menu of the documentation find commands and their arguments issued in the current session. You can reinstate a list of candidates by using this command.

This command is very helpful when, for instance, you need to cycle through several lists. Instead of reconstructing a candidate list each time you want to look at it, just use Select Candidate List and click on the list that you want to see.

This command is also available as [Select] in the Document Examiner command pane menu.

Show Overview

Prompts for a topic name. Shows an overview of the given topic. This overview includes the type (section or function, for instance) and name of the topic, possibly a short summary of the topic, the names of any other topics in the documentation that include this one, the names of any printed books that contain the topic, and the topic's keywords. The names of the topic(s) and book(s) are mouse sensitive.

The overview facility is a good way to explore the context in which a topic occurs. This is the online equivalent of looking in a printed book at the immediately surrounding pages for a topic. For example, when you do Find Whole Word Candidates on "register" you get 16 candidates. Then when you click middle on "sys:array-register" you get an overview that says this record appears in the section "Array Registers" and in the book "Reference Guide to Symbolics-Lisp". Now you can do Find Table Of Contents on "Array Registers" and get the following listing:

Array Registers
  sys:array-register
  Array Registers and Performance
  Hints for Using Array Registers
  Array Register Restrictions

Using the overview facility in combination with Find Table Of Contents is analogous to turning pages back and forth to see what the context is for a particular topic. If the overview for a particular topic does not give you enough information, try an overview on something listed in the first overview. So, in the example above, you do an overview on "Array Registers" and find that it appears in the section "Arrays" and in the book "Reference Guide to Symbolics-Lisp". Now do a Find Table Of Contents on "Arrays" and you get a very long listing of that topics table of contents.

This command is also available as [Show (M)] in the Document Examiner command pane menu.

The following is an example of the overview for the function **string-lessp**:

Overview
Function: **string-lessp**
It is included in topic: "String Comparisons Ignoring Case, Style, and Bits"
It appears in documents: *"Release 6.0 Release Notes"*,*"Reference Guide to Symbolics-Lisp"*
Keywords: STRING LESSP

Lookup Commands Available in an Editor

What Document (m-X)

Displays the name of the printed book that contains the given documentation topic. If the topic is included in more than one book, the titles of all the books containing the given topic are listed. In the Document Examiner, this information is available in the topic overview by clicking middle on any mouse-sensitive item in the viewer or any item in the list of current candidates or list of bookmarks.

Lookup Commands Available At a Lisp Listener and in Zmacs

When you are typing at a Lisp Listener or in Lisp Mode in Zmacs, you can use the following input editor commands to look up the documentation for the current Lisp object. For example, pressing `m-sh-A` after typing `(login 'whit` displays the documentation for **login**. "Current", refers to the Lisp object that precedes point.

`m-sh-A`	Looks up the documentation for the current function.
`m-sh-V`	Looks up the documentation for the current variable.
`m-sh-F`	Looks up the documentation for the current flavor.

When you look up documentation at a Lisp Listener using one of these input editor commands, the documentation appears on the screen, and the input editor then redisplays whatever you were typing.

When these commands do not find any documentation for the current function/variable/flavor in the documentation database, they check the object itself for a documentation string. If they find a documentation string, the string is displayed.

It should be noted that once a documentation string is displayed in this manner, the string has been installed as the documentation in your world. Thereafter, the display does not change if the documentation string is changed. In other words, this facility does not provide support for your putting new documentation into the documentation database.

12.4 Document Examiner Window

When you look at the Document Examiner window you see the following panes:

Pane	*Description*
Viewer	Displays documentation.
Current candidates	Displays the menu of topics that appeared the last time you used one of the documentation find commands.
Bookmarks	Displays a list of bookmarks, which are the names of topics displayed in the viewer or added to the list of bookmarks without being displayed.
Commands	Accepts commands at the prompt to the left and displays a menu of selected Document Examiner commands to the right.

12.4.1 Document Examiner Viewer

The large area on the left of the Document Examiner window is called the viewer. Documentation is displayed in the viewer. You can have multiple viewers, just as you can have multiple editor buffers. The viewer currently visible is called the *current viewer*. You can choose another viewer by using the command menu item [Viewer].

To view documentation topics in the Document Examiner viewer, you can do one of several things:

- Click on mouse-sensitive items in the viewer.
- Click on topics in the list of current candidates or the list of bookmarks. See the section "Document Examiner List of Current Candidates", page 131. See the section "Document Examiner List of Bookmarks", page 132.

- Use the Show Documentation command in the command pane. See the section "Show Documentation (an Overview)", page 122.
- Use [Show] in the command pane menu. See the section "Document Examiner Command Pane", page 133.

In the Document Examiner, when you select a topic for viewing, the topic is displayed at the end of the current viewer and the topic's name is added to the list of bookmarks. Topics chosen for display in the viewer are separated by horizontal lines.

When you select a topic for viewing at a Lisp Listener or an editor, the topic is displayed there, added to end of the current Document Examiner viewer, and the topic name is added to the end of the list of bookmarks. However, when you abort out of viewing a topic at a Lisp Listener or an editor, the Document Examiner just adds the topic name to the end of the list of bookmarks and does not display the topic in the current viewer.

Examples of Lisp code whose lines are wider than the viewer display with those lines truncated. When you need to see such examples in their entirety, use the Command Processor command Show Documentation in a wider window (for example, a Lisp Listener).

In the viewer, cross-references and documented Lisp objects are mouse sensitive. The following actions can be performed on mouse-sensitive items:

Mouse click	*Action*
left	Displays the topic in the current viewer.
middle	Shows an *overview* of the topic, including the type (section or function, for instance) and name of the topic, possibly a short summary of the topic, the names of any other topic(s) in the documentation that include this one, the names of the printed books that contain the topic, and the topic's keywords. The names of the topic(s) and book(s) that include this topic are mouse sensitive. For example: **Overview** Function: **string-lessp** It is included in topic: "String Comparisons Ignoring Case, Style, and Bits" It appears in documents: *"Release 6.0 Release Notes","Reference Guide to Symbolics-Lisp"* Keywords: STRING LESSP Issuing any command, pressing any keyboard key, or clicking a mouse button causes this display to go away.
sh-middle	On a mouse-sensitive item in the viewer or list of current

	candidates, adds the name of the topic to the list of bookmarks. On an item in the list of bookmarks, discards the name of the topic from the list of bookmarks, and, if the topic has been displayed, discards the display from the current viewer.
right	Pops up a menu of several commands with which to act on the display. Commands listed but not mouse-sensitive do not apply to the pane on which you clicked.

You can create, remove, and hardcopy viewers whenever you want and select another viewer by using the following commands in the Document Examiner.

Command	*Action*
Select Viewer	Selects or creates a viewer, prompting for a name. Also available as [Viewer] in the command menu.
Remove Viewer	Removes a viewer, prompting for a name, then selects the last viewer displayed. Also available as [Viewer (M)] in the command pane.
Hardcopy Viewer	Prompts for the name of a viewer and prints that viewer on the local Symoblics LGP. Also available as [Viewer (R)] in the command menu.

12.4.2 Document Examiner List of Current Candidates

The upper right-hand pane of the Document Examiner window contains the list of current candidates, the menu of topics that appeared the last time you used one of the documentation find commands. This menu remains until it is superseded by the next such command. It should be noted that lines that are wider than the list of current candidates pane are truncated.

You can reinstate a list of candidates by using the command Select Candidate List or the menu command [Select], which pops up a menu of the documentation find commands and their arguments issued in the current session.

The following actions can be performed on topics in the list of current candidates:

Mouse click	*Action*
left	Displays the topic in the current viewer.
middle	Shows an *overview* of the topic, including the type (section or function, for instance) and name of the topic, possibly a short summary of the topic, the names of any other topic(s) in the documentation that include this one, the names of the printed books that contain the topic, and the topic's keywords. The names of the topic(s) and book(s) that include this topic are mouse sensitive. For example:

	Overview Function: **string-lessp** It is included in topic: "String Comparisons Ignoring Case, Style, and Bits" It appears in documents: *"Release 6.0 Release Notes"*,*"Reference Guide to Symbolics-Lisp"* Keywords: STRING LESSP Issuing any command, pressing any keyboard key, or clicking a mouse button causes this display to go away.
sh-middle	On a mouse-sensitive item in the viewer or list of current candidates, adds the name of the topic to the list of bookmarks. On an item in the list of bookmarks, discards the name of the topic from the list of bookmarks, and, if the topic has been displayed, discards the display from the current viewer.
right	Pops up a menu of several commands with which to act on the display. Commands listed but not mouse-sensitive do not apply to the pane on which you clicked.

12.4.3 Document Examiner List of Bookmarks

The lower right-hand pane of the Document Examiner window contains the list of bookmarks. This is a history of bookmarks you place in the documentation. A bookmark is a pointer to a documentation topic. Each time you display a topic, a bookmark is placed in that topic, and the name of the topic is added to the list of bookmarks. You can also simply place a bookmark in a topic without displaying it in the viewer by clicking middle twice on an item in the list of current candidates. When you select another viewer, the list of bookmarks associated with it is also selected.

The list of bookmarks distinguishes between bookmarks whose topics have been displayed and those that have not. Topics that are displayed in the viewer are listed on a white background in the order in which you looked them up. Topics not displayed in the viewer follow and are listed on a gray background in the order in which you created the bookmarks. A marker on the list of bookmarks indicates the topic currently being displayed at the top of the viewer.

Lines that are wider than the list of bookmarks pane are truncated.

The following actions can be performed on topic names:

Mouse click	*Action*
left	Displays the topic in the current viewer.
middle	Shows an *overview* of the topic, including the type (section or function, for instance) and name of the topic, possibly a short summary of the topic, the names of any other topic(s) in the

documentation that include this one, the names of the printed books that contain the topic, and the topic's keywords. The names of the topic(s) and book(s) that include this topic are mouse sensitive. For example:

Overview
Function: **string-lessp**
It is included in topic: "String Comparisons Ignoring Case, Style, and Bits"
It appears in documents: *"Release 6.0 Release Notes","Reference Guide to Symbolics-Lisp"*
Keywords: STRING LESSP

Issuing any command, pressing any keyboard key, or clicking a mouse button causes this display to go away.

sh-middle	On a mouse-sensitive item in the viewer or list of current candidates, adds the name of the topic to the list of bookmarks. On an item in the list of bookmarks, discards the name of the topic from the list of bookmarks, and, if the topic has been displayed, discards the display from the current viewer.
right	Pops up a menu of several commands with which to act on the display. Commands listed but not mouse-sensitive do not apply to the pane on which you clicked.

12.4.4 Document Examiner Command Pane

The bottom portion of the Document Examiner window contains the command pane. The command pane offers completion on command names as well as topic names. c-? displays a mouse-sensitive list of possible completions. Press HELP to see a display of the available commands.

The command pane contains a command menu at the lower right. Use the following mouse clicks to perform these actions or commands.

Mouse	*Command*
[Help]	Brief command summary
[Help (M)]	Show the Document Examiner documentation
[Show]	Show Documentation
[Show (M)]	Show Overview
[Show (R)]	Find Table Of Contents
[Viewer]	Select Viewer
[Viewer (M)]	Remove Viewer
[Viewer (R)]	Hardcopy Viewer

[Find]	Find Whole Word Candidates (XXXXX)
[Find (M)]	Find Initial Substring Candidates (XXX....)
[Find (R)]	Find Any Candidates (..XXX..)
[Select]	Select Candidate List
[Private]	Read Private Document
[Private (M)]	Load Private Document
[Private (R)]	Save Private Document

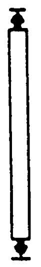
12.5 Repositioning Text in the Document Examiner

The Document Examiner viewer, list of current candidates, and list of bookmarks each have a bar located at its right edge. Each of the bars has a thick arrow at its top and bottom. These bars are one means for repositioning text in the viewer or items in the list of current candidates or list of bookmarks. The Document Examiner positioning mechanism is neither MORE processing nor scrolling. The positioning actions are more closely analogous to those positioning actions available in an editor. The Document Examiner positioning actions are described in the mouse documentation line when you move the mouse into one of these bars.

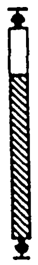
When you display a multipage topic, the positioning mechanism knows about only the part of the topic you have seen. If you look at only one page of a multi-page topic, the Document Examiner knows about only that page. Positioning in the Document Examiner works this way so as to limit the amount of space that documentation takes up in memory.

For instance, suppose you have "Arrays and Strings", a multipage topic, in a viewer followed by "setq". You have looked at only the first two pages of "Arrays and Strings". When you reposition backward from "setq", what you see is not the end of the topic "Arrays and Strings" but the last part of that topic that you have looked at. When you then reposition forward again, you see the next page of "Arrays and Strings" rather than "setq".

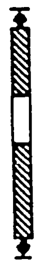
A bar looks like this when its pane is empty:



When the pane contains some text, part of the bar is gray. The part of the bar left blank indicates the percentage of the text currently displayed. The bar looks like this when its pane contains some text and the top 25 percent of that text is currently displayed:



The bar looks like this when its pane contains some text and the middle 25 percent of that text is currently displayed:



The mouse cursor appears differently and the positioning options change depending on the cursor's location in the bar.

Cursor	*Location*
⇧	Top bar arrow
⇩	Bottom bar arrow
⬆	Top gray section of bar
⬇	Bottom gray section of bar
⬌	Blank box

You can perform several types of positioning with the mouse and the bar. You can get a listing of positioning commands by pressing HELP while in the Document Examiner.

- To reposition text forward one screen:
  - Use c-V.
  - Use SCROLL.
  - Click left in the gray area near the bottom of the bar.

- To reposition text backward one screen:
  - Use m-V.
  - Use m-SCROLL.
  - Click left in the gray area near the top of the bar.

- To reposition quickly to any part of the current viewer:

  - Click left and hold the left button down on the blank box in the bar. Then drag the box to another position in the bar and the display changes accordingly.

- To reposition a few lines forward:
  - Use c-SCROLL.
  - Click left in the arrow at the bottom of the bar.

- To reposition a few lines backward:
  - Use c-m-SCROLL.
  - Click left in the arrow at the top of the bar.

- To reposition to the beginning or end of the current viewer or topic:
  - Use m-< to reposition to the beginning of the current viewer.
  - Use m-> to reposition to the end of the current viewer. (Please note that using this command while you have a partially displayed topic exposed in the viewer might cause an error. This is a known bug. To recover, simply press ABORT.)
  - Use the command Beginning of Topic to reposition to the beginning of the current topic.
  - Use the command End of Topic to reposition to the last screen dislayed for the current topic.

- To reposition the current screen:
  - Click middle next to a line to position it at the top of the screen.
  - Click sh-middle next to a line to position the top line at that place on the screen.
  - Click right next to a line to position it at the bottom of the screen.

12.6 Document Examiner Private Documents

The Document Examiner provides mechanisms for placing bookmarks in the online documentation and for creating private documents out of these bookmarks. The bookmarks are pointers to documentation topics in the database. A private document is a collection of bookmarks you put together and write out to a file. For instance, you might want a private document that consists of a set of documentation topics to which you frequently refer.

To create a private document you first create a list of bookmarks, either by looking up some topics or by clicking appropriately on the list of candidates or on mouse-sensitive items in the viewer. Then you save the list of bookmarks, using the command Save Private Document, answering its prompt with a pathname of a file to contain the bookmarks. You can load (Load Private Document) or read (Read Private Document) the private document back into the Document Examiner at any time, again answering the prompt with the pathname of the file that contains the bookmarks for the private document. For example:

1. Create a list of bookmarks consisting of some topics documenting login procedures and functions. For example:

   ```
   login-forms
   login-setq
   System Initialization Lists
   login
   ```

2. Use Save Private Document. The command prompts you and you answer with the pathname of a file to contain the private document's bookmarks. Following is the prompt for Save Private Document:

   ```
   Enter a pathname for the document to contain these bookmarks
   (default ACME-BLUE: /usr2/whit/private.psb):
   ```

   Pathname merging is supported by this command and the default location for a private document is always your home directory. So, with a home directory of /usr2/whit/ on ACME-BLUE, if you give Save Private Document the filename "login-book", the command writes the list of bookmarks to ACME-BLUE: /usr2/whit/login-book.psb.

The following commands manipulate private documents:

Save Private Document
: Saves the current list of bookmarks as a private document, prompting for a pathname. Save Private Document writes the list of bookmarks to the file whose pathname is given.

Read Private Document
: Reads a private document into your computer and shows it in the viewer. This command prompts for the pathname of a file containing the bookmarks of a private document and the name of a viewer to show it in. The default location for a private document is always your home directory and pathname merging follows the standard rules.

Load Private Document
: Loads a private document into your computer but does not show it in the viewer. This command prompts for the pathname of a file containing the bookmarks of a private document and the name of a viewer. The default location for a private document is always your home directory and pathname merging follows the standard rules.

Hardcopy Private Document
: Prints a private document on the local Symbolics LGP. This command prompts for the pathname of a file containing the bookmarks of a private document. The default location for a private document is always your home directory and pathname merging follows the standard rules.

13. Index of Function Keys

13.1 Introduction

This is a quick reference guide to the Symbolics 3600 Family function keys. Most of these keys have the same function in any window. However, a few of them perform differently in different contexts.

13.2 ABORT

When this is read by a program, the program aborts what it is doing and returns to its "command loop". Lisp Listeners, for example, respond to ABORT by throwing back to the read-eval-print loop (top level or **break**). Note that ABORT takes effect when it is read, not when it is pressed; it will not stop a running program.

13.3 c-ABORT

Aborts the operation currently being performed by the process you are typing at, immediately (not when it is read). For instance, this will force a Lisp Listener to abandon the present computation and return to its read-eval-print loop.

13.4 m-ABORT

When this is read by a program, the program aborts what it is doing and returns through all levels of commands to its "top level". Lisp Listeners, for example, throw completely out of their computation, including any **break** levels, then start a new read-eval-print loop.

13.5 c-m-ABORT

A combination of c-ABORT and m-ABORT, this immediately throws out of all levels of computation and restarts the process you type at which you type it.

13.6 BACKSPACE

In a Lisp Listener, BACKSPACE moves the cursor back, as does c-B, so that you can insert additional text or edit. In Zmacs, Converse, and Zmail message windows, it inserts a BACKSPACE into the buffer. In the main Zmail window it scrolls the current message backward (as does m-SCROLL).

13.7 CLEAR INPUT

Usually erases the input expression you are typing.

13.8 COMPLETE

Completes partially typed commands.

13.9 END

Marks the end of input to many programs. Single-line input can be ended with RETURN, but END will terminate multiple-line input where RETURN is useful for separating lines. The END key does not apply when typing in Lisp expressions, which are self-delimiting. END terminates input you have edited: See the section "Using the Input Editor".

13.10 ESCAPE

Displays the input editor history. c-ESCAPE displays the global kill history. Sends Escape/Altmode (octal 033) in the Terminal program.

13.11 FUNCTION **Key**

This key is a prefix for a family of commands relating to the display, which you can type at any time, no matter what program you are running.

13.11.1 Display and Hardcopy Commands

The FUNCTION commands that control screen display and hardcopying are:

RUBOUT	Does nothing; press this key to cancel FUNCTION if you typed the latter by accident.
CLEAR INPUT	Discards typeahead.
REFRESH	Clears and redisplays all windows.
A	Arrests the process shown in the status line. FUNCTION - A resumes the process.
B	Buries the currently selected window, if any — that is, it moves it underneath all other windows. This usually brings up some other window, which is automatically selected.
C	Complements the entire screen. An argument of 1 means white-on-black; an argument of 0 means black-on-white.
c-C	Complements the selected window, with the same argument as FUNCTION C.
m-C	Complements the mouse documentation line, with the same argument as FUNCTION C.
F	Shows users logged in on the associated machine. With numeric arguments, it shows users logged in on various machines.
H	Shows status of network hosts. With an argument, it prompts for hosts.
M	Controls global MORE processing. No argument means toggle, 0 means turn off, 1 means turn on.
c-M	Controls MORE processing for the selected window. The arguments are the same as for FUNCTION M.
O	Selects another exposed window.
Q	Hardcopies the entire screen.
c-Q	Hardcopies the selected window.
m-Q	Hardcopies the entire screen, minus the status and mouse documentation lines.

13.11.2 Selection and Notification Commands

The `FUNCTION` commands that control window selection and notification are:

`S`	Selects the most recently selected window. With an argument *n* (default is 2), it selects the *n*th previously selected window and rotates the top *n* windows. An argument of 1 rotates through all windows (a negative argument rotates in the other direction); 0 selects a window that requires attention (for example, to report an error).
`T`	Controls the selected window's input and output notification characteristics. If an attempt is made to output to a window when it is not exposed, one of three things can happen: the program can simply wait until the window is exposed, it can send a notification that it wants to type out and then wait, or it can quietly type out "in the background"; when the window is next exposed the output will become visible. Similarly, if an attempt is made to read input from a window that is not selected (and has no typed-ahead input in it), the program can either wait for the window to become selected, or send a notification that it wants input and then wait.

The `FUNCTION T` command controls these characteristics based on its numeric argument, as follows:

no argument	If output notification is off, turns input and output notification on. Otherwise turns input and output notification off. This essentially toggles the current state.
0	Turns input and output notification off.
1	Turns input and output notification on.
2	Turns output notification on, and input notification off.
3	Turns output notification off, and input notification on.
4	Allows output to proceed in the background, and turns input notification on.
5	Allows output to proceed in the background, and turns input notification off.

`W`	Controls the status line. With no argument, the status line is redisplayed. The numeric arguments control what process the status line watches. The options are:

0	Gives a menu of all processes, and freezes the status line on the process you select. When the status line

	is frozen on a process, the name of that process appears where your user ID normally would (next to the date and time), and the status line does not change to another process when you select a new window.
1	The status line watches whatever process is talking to the keyboard, and changes processes when you select a new window. This is the default initial state.
2	Changes the status line so that it displays the name of the process instead of the name of the user. This also freezes the status line on that process; normally the status line switches to display a different process whenever the window system tells it to. Use this if you see an unexpected state in the status line. It will help you find out what process is in that state; you may find that you are not talking to the process you think you should be.
3	Rotates the status line among all processes.
4	Rotates the status line in the other direction.

13.11.3 Recovering From Stuck States

The following FUNCTION commands should all be used with caution.

ESCAPE	Helps you recover from stuck states such as "Output Hold" and "Sheet Lock".
c-A	Arrests all processes except the one shown in the status line and critical system processes, such as the keyboard and mouse processes. FUNCTION - c-A resumes all processes arrested by this command.
SUSPEND	Gets to the cold-load stream.
c-T	Deexposes temporary windows. This is useful if the system seems to be hung because there is a temporary window on top of the window that is trying to type out.
c-CLEAR-INPUT	Clears window system locks. This is a last resort, although not as drastic as warm booting. Use this when none of the windows will talk to you, when you cannot get a System menu, and so on.

13.12 HELP

Usually gets you some online documentation or programmed assistance.

13.13 LINE

The function of this key varies considerably. It is used as a command by the Debugger, and sends a line feed character in the Terminal program.

13.14 LOCAL

On 3670 and 3640 consoles this key controls local console functions:

LOCAL-G	Rings the bell.
LOCAL-D	Makes the screen dimmer.
LOCAL-B	Makes the screen brighter.
LOCAL-Q	Makes the audio quieter.
LOCAL-L	Makes the audio louder.
LOCAL-*n* LOCAL-C	Changes the contrast of the screen. *n* is a digit between 1 and 4.

13.15 NETWORK

This key is used to get the attention of the Terminal program. As such it functions as a command prefix. You must be connected to a host via the Terminal program before you can use this key.

13.16 PAGE

In Zmacs (in searches and after c-Q) this key inserts a page separator character, which displays as "page" in a box.

13.17 REFRESH

Usually erases and redisplays the selected window.

13.18 RESUME

Continues from the **break** function and the Debugger. In the Terminal program this sends a backspace character.

13.19 RETURN Key

"Carriage return" or end of line. The exact significance of carriage return varys according to context.

13.20 RUBOUT

Usually erases the last character typed.

13.21 SCROLL

In Zmail and the Document Examiner, scrolls the display forward. m-SCROLL scrolls it backward.

13.22 SELECT Key

This key is a prefix for a family of commands, generally used to select a window of a specified type, such as a Lisp Listener or Zmail. The current list is:

λ (sy-sh-L)	Common Lisp
C	Converse
D	Document Examiner
E	Editor
F	File system maintenance
I	Inspector
L	Lisp
M	Zmail
N	Notifications
P	Peek
T	Terminal
X	Flavor Examiner

c-SELECT creates a new window of the specified type.

13.23 SUSPEND

Usually forces the process you are typing at into a **break** read-eval-print loop, so that you can see what the process is doing, or stop it temporarily. The effect occurs when the character is read, not immediately. Press RESUME to continue the interrupted computation (this applies to the three modified forms of the SUSPEND key as well).

13.24 c-SUSPEND

This is like SUSPEND, but takes effect immediately rather than when it is read. Press RESUME to continue the interrupted computation.

13.25 m-SUSPEND

Forces the process you type it at into the Debugger when it is read. It should type out ">>BREAK:", any proceed options, and the Debugger prompt "→". You can examine the process, then press RESUME or c-C to continue.

13.26 c-m-SUSPEND

Forces the process you type it at into the Debugger, whether or not it is running.

13.27 SYMBOL

Acts as a modifier key to produce special characters. Pressing sym-HELP produces a display of special function and special character keys.

13.28 TAB Key

This key is only sometimes defined. Its exact function depends on context, but in general it is used to move the cursor to an appropriate point to the right.

13.29 Keys Not Currently Used

The following keys currently have no function:

```
MODELOCK
REPEAT
```

The following keys are reserved for use by the user (for example, to put custom editor commands or keyboard macros on):

```
CIRCLE
SQUARE
TRIANGLE
HYPER
```

14. Quick Summary of Mouse Functions

14.1 Mouse Cursor Shape

These are some of the more common mouse cursors:

A thin arrow pointing North by Northwest (up and to the left). This is the default mouse cursor. It indicates that there no special commands on the mouse buttons. Clicking left will select the window pointed to. Clicking right will get you the System menu.

A thin arrow pointing North by Northeast (up and to the right). This means the mouse is in an editor window. You have several editor commands on the mouse buttons. See the section "Mouse Documentation Line in Zmacs" in *Text Editing and Processing*.

A slightly thicker arrow pointing North (straight up). The editor uses this to show that it is asking you for the name of a function or for a symbol. If you point the mouse at a function name, and stop moving it, the name will light up and you can click to select it.

A small x. This is used when the mouse cursor should be unobtrusive, for instance in menus.

14.2 Scrolling with the Mouse

Some windows display "contents" that are too big to fit entirely in the window. The editor and the inspector are examples. When this is the case, you see only a portion of the contents, and you can scroll the contents up and down using the mouse.

The following mouse cursors indicate that the mouse is being used to control scrolling:

- A fat arrow, pointing up or down. This indicates you are in a scrolling zone. Moving the mouse slowly in the direction of the arrow scrolls the window, revealing more of the text in the direction the arrow points.

- Scrolling zones often say *more above* or *more below* in small italic letters. Clicking on one of these legends scrolls the window up and down by its height, thus you see the next or previous windowful. When the top or bottom of the window contents is reached, so that it is not possible to scroll any farther in one direction, the legend in the scrolling zone changes to indicate this.

- A fat double-headed arrow. A thin black bar appears at the left border of the window, the "scroll bar". The size of this bar relative to the edge of the window to which it is attached shows what portion of the window's contents is visible. The vertical position of the bar within the edge shows the position of the visible portion of the window's contents relative to the whole. The mouse commands in this case are:

Left	Move the line next to the mouse to the top of the window.
sh-Left	Move the line next to the mouse to the bottom of the window.
Right	Move the top line to where the mouse is.
sh-Right	Move the bottom line to where the mouse is. Because of this command definition, you cannot get to the System menu while the mouse is displaying a double-headed fat arrow.
Middle	Jump to a place in the window's contents as far, proportionally, from its beginning as the mouse is from the top of the window.

15. A Brief Introduction to the Lisp World

15.1 The Lisp Top Level

These functions constitute the Lisp top level and its associated functions.

si:lisp-top-level *Function*

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 &optional (*called-by-user* **t**) *Function*

This function restarts the Lisp system, resetting the values of various global constants and initializing the error system.

si:lisp-top-level1 *stream* *Function*

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.

defvar-resettable *name initial-value* &optional (*warm-boot-value* **nil** *wbv-p*) *documentation* *Special Form*

defvar-resettable is like **defvar**, except that it also maintains a *warm-boot value*. See the section "Variables" in *Reference Guide to Symbolics-Lisp*. During a warm-boot, the system sets the variable to its warm-boot value. If you want a variable to be reset at warm boot time, define it with **defvar-resettable**.

15.1.1 Standard Variables

Standard variables are special variables that are used to control some aspect of the Lisp environment. These are the currently defined standard variables, their standard values, and their valid values:

symbol	*standard value*	*valid values*
base	10	2 ≤ base ≤ 36
ibase	10	2 ≤ base ≤ 36
***nopoint**	**t**	
prinlevel	**nil**	**nil** or a non-negative integer
prinlength	**nil**	**nil** or a non-negative integer
si:*princase*	**:upcase**	**:upcase, :downcase**

		or **:capitalize**
si:prinradix	**nil**	**nil** or 2 ≤ base ≤ 36
prinl	**nil**	
si:*print-gensym*	**t**	
prinarray	**nil**	
package	**si:pkg-user-package**	[1]
readtable	**si:standard-readtable**	[2]
default-cons-area	**working-storage-area** [3]	
cl:*read-default-float-format*		**'cl:single-float**
fs:*automatically-login-to-sys-host*		**nil**
neti:*inhibit-obsolete-information-warning*		**nil**
alphabetic-case-affects-string-comparison		**nil**

Notes:

1. The value of **package** must be an unlocked package that uses one of the packages in **si:*reasonable-packages***.

2. The **readtable** must be one of the readtables on the list **si:*valid-readtables***.

3. The value of **default-cons-area** must be an allocated area.

The following functions and variables pertain to standard variables:

defvar-standard *name initial-value* &optional (*warm-boot-value* **nil** *wbv-p*) (*standard-value* **nil** *sv-p*) *validation-predicate documentation* *Special Form*

defvar-standard is like **defvar-resettable**, except that it also defines a *standard value* that the variable should be bound to in command and breakpoint loops. For example, the standard values of **base** and **ibase** are 10. The *validation-predicate* is used to ensure that the value of the variable is valid when it is bound in command loops.

base is defined like this:

```
(defvar-standard base 10. 10. 10. validate-base)
(defun validate-base (b)
   (and (fixnump b) (< 1 b 37.)))
```

setq-standard-value *name form* &optional (*setq-p* **t**) (*globally-p* **t**) (*error-p* **t**) *Special Form*

setq-standard-value sets the standard value of *name* to the value of *form*. If you want to change your default **base** to 8 (octal), do this:

```
(setq-standard-value base 8)
(setq-standard-value ibase 8)
```

setq-standard-value runs the validation function associated with the symbol and signals an error if the validation function fails. You can only use **setq-standard-value** on symbols defined with **defvar-standard**.

When a breakpoint of some kind is entered, the system finds out the standard values for all the symbols defined with **defvar-standard**. It then compares these values against the current bindings for these symbols. If the current bindings do not match the standard bindings, you are warned, and the symbols are bound to the standard values.

There are two association lists, **si:*standard-bindings*** and **si:*interactive-bindings***. **si:*interactive-bindings*** is never set, only bound and **si:*standard-bindings*** is never bound, only set. The standard binding for a variable is gotten by looking on **si:*interactive-bindings***. If no binding is found on **si:*interactive-bindings***, then **si:*standard-bindings*** is checked. For example, **zwei:com-break** puts the value of **package**, **base**, and **ibase** from the file attribute list onto **si:*interactive-bindings*** so that they become the standard binding for Zmacs. **pkg-goto** puts the new value of **package** onto **si:*standard-bindings***. Evaluation of forms in Zmacs, for example, Evaluate Into Buffer m-X, also binds the symbols to their standard values.

As a result, whenever you enter a breakpoint you are guaranteed predictable, consistent behavior with regard to the bindings of these variables.

standard-value-let *vars-and-vals* &body *body* *Macro*

Like **let** except that it also pushes the values in *vals* onto the **si:*interactive-bindings*** alist, causing them to become the new standard bindings. All the symbols in *vars* are then bound to *vals* (with a **let**) and *body* is executed in this context.

Example:

```
(defun octal-top-level ()
  (standard-value-let
    ((base 8)
     (ibase 8)
     (package (pkg-find-package 'new-command-loop)))
    (let ((standard-io 'terminal-io))
        (loop
             as form = (read)
             do (print (eval form))))))
```

standard-value-let* *vars-and-vals* &body *body* *Macro*

Like **let*** except that it also pushes the values in *vals* onto the **si:*interactive-bindings*** alist, causing them to become the new standard bindings. All the symbols in *vars* are then bound to *vals* (with a **let***) and *body* is executed in this context.

standard-value-progv *vars vals* &body *body* *Macro*

Causes all of the symbols in *vars* to have their corresponding value in *vals* pushed onto the **si:*interactive-bindings*** alist (that is, those values become the new standard bindings). All the symbols in *vars* are then bound to *vals* (with a **progv**) and *body* is executed in this context. This is useful for writing Lisp style command loops.

si:standard-readtable *Variable*

The value of **si:standard-readtable** is that **readtable** to use when typing forms interactively to the Lisp interpreter. When a distribution world is cold booted, the value of **si:standard-readtable** is a copy of **si:initial-readtable**. If you wish to customize the syntax of forms typed to the Lisp interpreter, you should make your customizations to **si:standard-readtable**. **readtable** is bound to **si:standard-readtable** whenever a break loop or debug loop is entered. **readtable** is set to **si:standard-readtable** using the standard variable mechanism whenever the machine is warm booted.

If warm booting the machine were impossible, then **si:standard-readtable** would not be necessary. The top-level value of **readtable** could be used instead. However, if the machine is warm booted while **readtable** is bound, the top-level value of **readtable** is lost.

Examples:

- This example illustrates the use of binding **readtable** in order to implement a special syntax. Forms are to be read from a file while preserving the case of symbols.

```
(defvar case-sensitive-readtable (copy-readtable))

(loop for code from (char-code #/a) to (char-code #/z)
      as char = (code-char code)
      do (setf (si:rdtbl-trans case-sensitive-readtable code) char))

(defun read-case-sensitive-file (file)
  (with-open-file (stream file :direction :input)
    (let ((readtable case-sensitive-readtable))
      (loop do (process-form (read stream))))))
```

  In case an error occurs while inside **process-form** or inside a reader macro invoked by **read, readtable** is bound to **si:standard-readtable,** which is most useful for debugging.

- This example illustrates the use of **si:standard-readtable** and **si:initial-readtable** to customize the environment for typing expressions interactively. "@" is defined as an abbreviation for **location-contents**, in the same manner that "'" is an abbreviation for **quote**.

```
(defun at-sign-macro (ignore stream)
  (values (list 'location-contents (read stream)) 'list))

(defvar my-readtable (copy-readtable))
(set-syntax-macro-char #/@ 'at-sign-macro my-readtable)

(defun enable-my-readtable ()
  (setq si:standard-readtable my-readtable)
  (setq readtable my-readtable))

(defun disable-my-readtable ()
  (setq si:standard-readtable si:initial-readtable)
  (setq readtable si:initial-readtable))
```

While it is useful for the user to set the values of **readtable** and **si:standard-readtable**, the value of **si:initial-readtable** should never be changed. In addition, the **readtable** that is the value of **si:initial-readtable** should never be modified, modifications should be made only to the **readtable** that is the value of **si:standard-readtable**. See the function **copy-readtable** in *Reference Guide to Symbolics-Lisp.*

See the section "The Readtable" in *Reference Guide to Symbolics-Lisp.*

prinl *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 **prinl** 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 **prinl** function or any of its relatives such as **print** and **format**; to do that, you need more information on customizing the printer. See the section "Output Functions" in *Reference Guide to Streams, Files, and I/O.* If you set **prinl** 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.

15.2 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 that is loaded when you log in; you can use it 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. Thus, for every form in the init file, you should add to the list that is the value of **logout-list** a Lisp form to undo its effects. The functions **login-forms** and **login-setq** help make this easy.

login *user-name* &key *host* (*load-init-file* **t**) *no-query-when-unknown* *Function*

login logs the user *user-name* in to the Lisp Machine. *host* is a particular host computer. If the value of *load-init-file* is **t**, as it is by default, the user's init file is loaded from *host*. If the value of *load-init-file* is **nil** the init file is not loaded. This is the Lisp version of the Login command in the Command Processor. See the section "Login Command", page 28.

The user object in the namespace database that contains your *user-name* also contains the name of the host(s) where your mail and init files reside. Therefore, you seldom need to supply a *host* argument to **login**.

You can log in as a registered user by not specifying a host, or you can log in to a specific host as a user on that host, not registered in the Lisp Machine namespace database.

If *host* requires passwords for logging in, you are asked for a password. When logging in to a TOPS-20 host, typing an asterisk before your password enables any special capabilities you might be authorized to use.

If anyone is logged in to the machine already, **login** logs that user out before logging in *user-name*. See the function **logout**, page 157. **login** also runs the **login-initialization-list**. See the section "System Initialization Lists" in *Internals, Processes, and Storage Management*.

When **login** loads an init file, it looks for a file whose name depends on the host. See the section "Init File Naming Conventions" in *Reference Guide to Streams, Files, and I/O*. Init files should be written using **login-forms** so that **logout** can undo them. Usually, however, you cold boot the machine before logging in, to remove any traces of the previous user.

A typical use of **login** looks like this:

```
(login 'kjones)
```

A typical use of **login** to avoid loading your init file looks like this:

```
(login 'kjones :load-init-file nil)
```

Supplying all the arguments might look like this:

```
(login 'kjones :host local :load-init-file nil)
```

The host `local` is particularly useful if your namespace system is down and you wish to log in to your Lisp Machine without having it try to use the namespace database.

If you supply an unknown user id and do not specify **:host**, you are given an opportunity to specify a particular host for the current login session, and to add the user object thus created to the network database (accomplished via

Edit Namespace Object or **tv:edit-namespace-object**) for subsequent logins. You can instead select the Retry option, which is useful when the namespace server did not respond to your initial **login** request.

The **:no-query-when-unknown** keyword is for internal use by the system, it is not intended for users.

logout *Function*

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 and returns **t**. See the section "Initializations" in *Internals, Processes, and Storage Management.*

user-id *Variable*

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

site-name *Variable*

The value is a keyword, the name of the site at which this machine is located. **site-name** can be used to conditionalize programs. For example:

```
(when (eq site-name :acme)
  (load "apricot:>smith>cerebrum-server"))
```

Site names are described in more detail: See the section "Namespace System Site Objects" in *Networks.*

logout-list *Variable*

The value of **logout-list** is a list of forms that are evaluated when a user logs out.

login-forms &body *forms* *Special Form*

login-forms is a special form for wrapping around a set of forms in your init file. It evaluates the forms and arranges for them to be undone when you log out.

login-forms always evaluates the forms, even when it does not know how to undo them. For forms that it cannot undo, it prints a warning message.

In the following example, **login-forms** arranges for the base to be reset at logout to 10 (the default) and for **bar** either to become undefined or to get its old function definition. It would warn you about **quux** being impossible to undo.

```
(login-forms
        (setq-standard-value base 8)
        (setq-standard-value ibase 8)
        (defun bar (x y) (+ x y))
        (quux 3))
```

You can create functions to undo forms that **login-forms** does not recognize. To undo a given form, you put a property on the symbol that is the car of the form to undo. For example, to create a function to undo **quux**:

```
(defun (:property quux :undo-function) (form)
  `(undo-quux ,(cadr form)))
```

The value returned by an undo function is a form to be evaluated at logout time.

setq-globally &rest *vars-and-vals* *Special Form*

setq-globally should be used with **login-forms** for anything that might be bound while evaluating the **login-forms**.

setq-globally works like **setq** but sets the global values, bypassing any special-variable bindings. **login-forms** knows how to undo this. **setq-globally** is the recommended way to set things in your init file that are not set with **setq-standard-value**.

An example:

```
(login-forms
  (setq-globally zwei:*converse-beep-count* 4))
```

15.3 Some Utility Functions

zwei:save-all-files &optional (*ask* **t**) *Function*

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 command Save File Buffers (m-X) does, but it stays away from redisplay and other advanced facilities so that it might work if other things are broken.

zwei:zmail-save-all-files is similar, but saves mail files from Zmail.

ed &optional *thing* *Function*

ed is the main Lisp function for entering Zmacs. Select Activity Zmacs is the command for entering Zmacs.

(ed) or **(ed nil)** enters Zmacs, leaving everything as it was when you last left Zmacs. If Zmacs has not yet been used in the current session, it is initialized and an empty buffer created.

(ed t) enters Zmacs, and creates and selects an empty buffer.

If the argument is a pathname or a string, the **ed** function enters Zmacs, and finds or creates a buffer with the specified file in it. This is the same as the Edit File command.

If the argument is a symbol that is defined as a function, Zmacs will try to find the source definition for that function for the user to edit. This is the same as the Edit Definition command.

Finally, if the argument is the symbol **zwei:reload**, Zmacs will be reinitialized. All existing buffers will be lost, so use this only if you have to.

dired &optional (*pathname* "") *Function*

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 press the HELP key for further information. This is similar to the Edit Directory command, except that Edit Directory enters Zmacs and runs Dired (m-X).

mail &optional *initial-destination initial-body prompt initial-idx* *Function*
bug-report (*make-subject*
(memq zwei:*require-subjects* (quote (t :init))))
initial-subject

Sends mail by putting up a window in which you can compose the mail.

initial-destination is a symbol or string that is the recipient.

initial-body is a string that is the initial contents of the mail. If these are unspecified they can be typed in during composition of the mail. Press the END key to send the mail and return from the **mail** function.

prompt and *initial-idx* are used by programs, such as **bug**, that call **mail**. *prompt* is a string printed in the minibuffer of the mail window created by **mail**. *initial-idx* positions point in the mail window.

bug &optional (*system* **(quote zwei:lispm)**) *additional-body prompt* *Function*
point-before-additional-body (*make-subject*
(memq zwei:*require-subjects* (quote (t :init :bug))))
initial-subject

Reports a bug. This is the same as the Report Bug command. **bug** is like **mail** but includes information about the system version and what machine you are on in the text of the message.

system is the name of the faulty program (a symbol or a string). It defaults to **lispm** (the Lisp Machine system itself). 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.

additional-body is user-supplied text appended to the information supplied by the system.

prompt is text supplied by the system printed in the minibuffer of the mail window concerning the bug-mail you are sending.

point-before-additional-body is a position for point supplied by the system.

qsend &optional *destination message* *Function*

Sends interactive messages to users on other machines on the network.

destination is normally a string of the form "*name@host*", to specify the recipient. If you omit the *@host* part and just give a name, **qsend** looks at all of the Lisp Machines at your site to find any that *name* is logged into; if the user is logged into one Lisp Machine, it is used as the host; if more than one, **qsend** asks you which one you mean. If you leave out *destination* altogether, doing just **(qsend)**, Converse is selected as if you had pressed SELECT C.

message should be a string. If it is omitted, **qsend** asks you to type in a message. You should type in the contents of your message and press END when you are done.

The input editor is used while you type in a message to **qsend**. So you get some editing power, although not as much as with full Converse (since the latter uses Zwei). See the section "Using the Input Editor". This function predates Converse and is retained for compatibility.

16. Customizing Your Environment

16.1 What is Customizing?

When you load a file or set a variable (for example, specifying that your hardcopies are sent to a certain printer, changing the default font, or changing the appearance of the command prompt), you alter the default system behavior in your environment for the rest of the time you remain logged in. This type of per-session customization does not remain in effect in your machine after you log out or cold boot. If you load a file or set a variable for an intentionally temporary effect, this is fine.

However, if you decide that you want these changes to be put into effect every time you log in (permanently in your environment), you can save them in an *init file*, thereby instructing the system to automatically execute this sequence of commands every time you log in.

16.2 Init Files

An init file is a Lisp program that gets loaded when you log in; it can be used to set up a personalized environment. An init file contains only Lisp forms. The name depends on the type of file system it is stored on:

3600	lispm-init.lisp
UNIX 4.1	lispm-init.l
UNIX 4.2	lispm-init.lisp
VMS	lispmini.lsp
TOPS-20	lispm-init.lisp
ITS	*name* lispm

A simple init file consists primarily of the **login-forms** and the **setq** special forms. The **login-forms** special form evaluates forms in your init file and arranges for them to be undone when you log out. The **setq** special form sets the value of one or more variables.

Here is an example of a simple init file:

```
; -*- Mode: LISP; Package: USER; Lowercase: T; Patch-file: T -*-

(login-forms
  (setq si:*cp-prompt* 'si:arrow-prompt)

  zwei:
  (setq text-mode-hook 'auto-fill-if-appropriate)
```

```
    (setq si:local-finger-location
          (cond ((y-or-n-p "in your office? ")
                 "340 Domingo x562")
                (t (format t "~&Where are you? ")
                   (readline query-io)))))

  (si:set-default-hardcopy-device "Echo-Lake")
  (si:set-screen-hardcopy-device "Echo-Lake")
```

In this simple init file, the first **setq** changes the value of the variable that displays the command processor prompt from the default `Command:` to an arrow. The second **setq** specifies that the system automatically fill text that you type in any editor-based activity when appropriate. The third **setq** sets the value of the variable that reports your user ID and on what machine you are logged in to ask you when you log in whether you are in your office, and if not, where you are so that it can send that information to the network namespace database.

The rest of the init file contains two functions that set the default printer for the various commands that hardcopy files and for the FUNCTION Q Screen Hardcopy command.

Here is the description of **setq**:

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

Used to set the value of one or more variables. The first *value* is evaluated, and the first *variable* is set to the result. Then the second *value* is evaluated, the second *variable* is set to the result, and so on for all the variable/value pairs. **setq** returns the last value, that is, the result of the evaluation of its last subform. Example:

```
(setq x (+ 3 2 1) y (cons x nil))
```

x is set to **6**, **y** is set to **(6)**, and the **setq** form returns **(6)**. Note that the first variable was set before the second value form was evaluated, allowing that form to use the new value of **x**.

If you do not cold boot your machine after each session, you should arrange for your customizations to be undone when you log out. You do this by using **login-forms**:

login-forms &body *forms* *Special Form*

login-forms is a special form for wrapping around a set of forms in your init file. It evaluates the forms and arranges for them to be undone when you log out.

login-forms always evaluates the forms, even when it does not know how to undo them. For forms that it cannot undo, it prints a warning message.

In the following example, **login-forms** arranges for the base to be reset at logout to 10 (the default) and for **bar** either to become undefined or to get its old function definition. It would warn you about **quux** being impossible to undo.

```
(login-forms
        (setq-standard-value base 8)
        (setq-standard-value ibase 8)
        (defun bar (x y) (+ x y))
        (quux 3))
```

You can create functions to undo forms that **login-forms** does not recognize. To undo a given form, you put a property on the symbol that is the car of the form to undo. For example, to create a function to undo **quux**:

```
(defun (:property quux :undo-function) (form)
  `(undo-quux ,(cadr form)))
```

The value returned by an undo function is a form to be evaluated at logout time.

setq-standard-value is a special form, similar to **setq**, that you should use if you reset any of the variables that control aspects of the Lisp environment (for example, the default base) as opposed to convenience features. See the section "Standard Variables", page 151.

Other variables can be set inside **login-forms** using **setq-globally**:

setq-globally &rest *vars-and-vals* *Special Form*

setq-globally should be used with **login-forms** for anything that might be bound while evaluating the **login-forms**.

setq-globally works like **setq** but sets the global values, bypassing any special-variable bindings. **login-forms** knows how to undo this. **setq-globally** is the recommended way to set things in your init file that are not set with **setq-standard-value**.

An example:

```
(login-forms
  (setq-globally zwei:*converse-beep-count* 4))
```

To load individual files from your init file, use the **load** function:

```
(load "vixen://usr//kjones//tools//toolkit")
(load "SYS: LISP; MY-PROJECT")
(load "Tuna:>kjones>examples>decorate")
```

The first sample form loads a file from a Unix system in the appropriate syntax (the slashes are doubled). The second form loads a file using its logical pathname; the third form loads a Lisp Machine file using its physical pathname.

To load a *system* consisting of many files, use the **make-system** function:

```
(make-system 'my-tools :noconfirm :compile :silent)
```

This **make-system** form makes a system named my-tools, performing all necessary transformations without asking for confirmation (:noconfirm), loading the compiled code files or newest versions of source files (:compile), and suppressing reporting of each transformation as it occurs (:silent). See the section "Making a System" in *Program Development Utilities.*

16.3 How to Create an Init File

The easiest way to create an init file is by copying the sample init file shown here and then building on it, or by copying someone else's init file. Often you acquire customizations that you find out about from people who have been using Lisp Machines longer than you.

16.4 Logging in Without Processing Your Init File

You might want to log in and work in the standard default system environment, that is, without having your init file set up your usual customizations. Perhaps you want to test a program of yours in the standard environment or try a new system feature in an unpolluted environment. Log in this way:

`Login` *username* `:init file none`

to tell the Login command that you do not want your init file automatically loaded.

16.5 Customizing the Command Processor

You can change the command processor's mode, prompt, and special characters, and you can customize the display of the prompt and help messages. Usually you customize the command processor by setting special variables. You might want to do this in your init file, inside a **login-forms** special form.

Whenever you change the command processor's mode, prompt, or other characteristics, you set its state for all Lisp Listeners and **break** loops. You cannot put the command processor into one mode in one Lisp listener and another mode in another.

If you change the command processor's mode or prompt, or if you turn the command processor on or off, the change takes place immediately in that Lisp Listener or **break** loop but not in others that are waiting for input. For example, suppose you use the Set Command Processor command in a **break** loop to change the prompt and dispatch mode. These changes do not take effect in a Lisp Listener that is waiting for input until you execute a command or form or you press ABORT there.

16.5.1 Setting the Command Processor Mode

The command processor *mode* determines how input is treated. Following are the four modes and their meanings:

:form-only	All input is treated as a Lisp form.
:command-only	All input is treated as a command invocation.
:form-preferred	Input is treated as a Lisp form unless you precede it by a command dispatch character. In this case it is treated as a command invocation. By default, the command dispatch character is a colon.
:command-preferred	Input is treated as a command invocation if it begins with an alphabetic character. Input is treated as a Lisp form if it is does not begin with an alphabetic character or if you precede it by a form dispatch character. By default, the form dispatch character is a comma.

You can set the command processor mode for Lisp Listeners and **break** loops in two ways:

1. Use the Set Command Processor command. The first argument to this command is the dispatch mode. See the section "Set Command Processor Command", page 32.

2. Set the value of the special variable **si:*cp-dispatch-mode***.

si:*cp-dispatch-mode* *Variable*

The current command processor dispatch mode in Lisp Listeners and **break** loops; a keyword. Possible values are **:form-only**, **:form-preferred**, **:command-only**, and **:command-preferred**. For the meanings of these values: See the section "Setting the Command Processor Mode", page 165. The default is **:command-preferred**.

The default dispatch mode for **read-command-or-form** is the value of **si:*cp-default-dispatch-mode***.

16.5.2 Setting the Command Processor Prompt

You can set the command processor prompt for Lisp Listeners and **break** loops in two ways:

1. Use the Set Command Processor command. The second argument to this command is a string to be displayed as the prompt. See the section "Set Command Processor Command", page 32.

2. Set the value of the special variable **si:*cp-prompt***.

si:*cp-prompt* *Variable*

A prompt option for displaying the current command processor prompt in Lisp Listeners and **break** loops. The value of this variable is passed to the input editor as the value of the **:prompt** option. The value can be **nil**, a string, a function, or a symbol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor" in *Programming the User Interface*.

The default is "Command: ". If the value is **nil** or the empty string, no prompt is displayed. If the value is **si:arrow-prompt**, an arrow is displayed as the prompt.

The default prompt for **read-command** and **read-command-or-form** is the value of **si:*cp-default-prompt***.

16.5.3 Setting Command Processor Special Characters

You can change the command and form dispatch characters by setting the special variables **si:*cp-command-dispatchers*** and **si:*cp-form-dispatchers***.

si:*cp-command-dispatchers* *Variable*

A list of characters that precede commands, distinguishing them from input to the Lisp interpreter, when the command processor is in **:form-preferred** mode. The default is **(#/:)**.

si:*cp-form-dispatchers* *Variable*

A list of characters that precede Lisp forms, distinguishing them from commands, when the command processor is in **:command-preferred** mode. (These characters are needed only when the Lisp form begins with an alphabetic character.) The default is **(#/,)**.

16.5.4 Customizing Command Processor Display

By setting special variables, you can control the action the command processor takes when you type a blank line and how it displays the screen when you ask for help.

si:*cp-blank-line-mode* *Variable*

A keyword that determines what action the command processor takes when you type a blank line in Lisp Listeners and **break** loops:

:reprompt	Redisplay the prompt, if any. This is the default.

:beep	Beep.
:ignore	Do nothing.

The default blank line mode for **read-command** and **read-command-or-form** is the value of **si:*cp-default-blank-line-mode***.

si:*cp-typeout-default* *Variable*

A keyword that determines how the command processor prints help messages. Possible values are those acceptable as the first argument to the **:start-typeout** message to interactive streams:

:insert	The help message, like a notification, is inserted before the current input.
:overwrite	The help message is inserted before the current input, but the next time an **:insert** or **:overwrite** operation is done, this message is overwritten. This is the default.
:append	The help message appears after the current input, which is reprinted after the help message.
:temporary	The help message appears after the current input and disappears when you type the next character.
:clear-window	The window is cleared and the help message appears at the top.

For more information: See the method **(:method si:interactive-stream :start-typeout)** in *Programming the User Interface.*

16.6 Zmacs Customization in Init Files

You can set Zmacs parameters in your init file also. This section gives you some guidelines for how to set different types of parameters. For information about the available features: See the section "Zmacs Manual" in *Text Editing and Processing.*

16.6.1 Setting Editor Variables

The forms described show how to set Zmacs variables (the kind that Set Variable (m-X) sets).

To set these variables, which are symbol macros, you must use the **setf** macro. For a description of symbol macros: See the section "Symbol Macros" in *Reference Guide to Symbolics-Lisp.* For a description of the **setf** macro: See the macro **setf** in *Reference Guide to Symbolics-Lisp.*

16.6.1.1 Ordering Buffer Lists

```
(SETF ZWEI:*SORT-ZMACS-BUFFER-LIST* NIL)
```

This displays the list of buffers in the order the buffers were created rather than in the order they were most recently visited.

16.6.1.2 Putting Buffers Into Current Package

```
(SETF ZWEI:*DEFAULT-PACKAGE* NIL)
```

This puts buffers created with c-X B (Select Buffer) into whatever package is current; the default is to put them in the **user** package.

16.6.1.3 Setting Default Major Mode

```
(SETF ZWEI:*DEFAULT-MAJOR-MODE* ':TEXT)
```

This sets the default major mode to Text Mode for buffers with no Mode attribute and no major mode deducible from the file type; the default is Fundamental Mode.

16.6.1.4 Setting Find File Not to Create New Files

```
(SETF ZWEI:*FIND-FILE-NOT-FOUND-IS-AN-ERROR* T)
```

This beeps and prints an error message when you give c-X c-F (Find File) the name of a nonexistent file. The default prints `(New File)` and creates an empty buffer, which when saved by c-X c-S (Save File) creates the file that was nonexistent.

16.6.1.5 Setting Goal Column for Real Line Commands

```
(SETF ZWEI:*PERMANENT-REAL-LINE-GOAL-XPOS* 0)
```

This moves subsequent c-N and c-P (Down Real Line and Up Real Line) commands to the left margin, like doing c-0 c-X c-N (Set Goal Column to zero).

16.6.1.6 Fixing White Space for Kill/Yank Commands

```
(SETF ZWEI:*KILL-INTERVAL-SMARTS* T)
```

This tells the killing and yanking commands optimize white space surrounding the killed or yanked text.

16.6.2 Key Bindings

To bind keys, you first define the comtab in which to put the binding. For example, ***standard-comtab*** and ***standard-control-x-comtab*** define features of all Zwei-based editors; ***zmacs-comtab*** and ***zmacs-control-x-comtab*** define features that are Zmacs-specific.

16.6.2.1 Balanced Quotation Marks and Asterisks

```
    ZWEI:(SET-COMTAB *STANDARD-COMTAB*
                     '(#\m-/" COM-MAKE-/(/)
                       #\c-m-/" COM-MOVE-OVER/)
                       #\m-/* COM-MAKE-/(/)
                       #\c-m-/* COM-MOVE-OVER-/)
                       ))
```

This defines commands to insert balanced pairs of quotation marks or asterisks into the buffer. For example, you can type an asterisked special variable name as m-* FOO, which inserts *FOO* into the buffer, ensuring that one does not forget to type the trailing asterisk.

16.6.2.2 White Space in Lisp Code

```
    ZWEI:(SET-COMTAB *STANDARD-CONTROL-X-COMTAB*
                     '(#\SP COM-CANONICALIZE-WHITESPACE))
```

This defines c-X SPACE as a command that makes the horizontal and vertical white space around point (or around mark if given a numeric argument or immediately after a yank command) conform to standard style for Lisp code.

16.6.2.3 c-m-L on the SQUARE Key

```
    ZWEI:(SET-COMTAB *ZMACS-COMTAB*
                     '(#\SQUARE COM-SELECT-PREVIOUS-BUFFER))
```

This defines the SQUARE key to do the same thing as c-m-L. This key binding is placed in ***zmacs-comtab*** rather than ***standard-comtab*** since buffers are a feature of Zmacs, not of all Zwei-based editors.

16.6.2.4 Edit Buffers on c-X c-B

```
    ZWEI:(SET-COMTAB *ZMACS-CONTROL-X-COMTAB*
                     '(#\c-B COM-EDIT-BUFFERS))
```

This makes c-X c-B invoke Edit Buffers rather than List Buffers. This key binding is placed in ***zmacs-control-x-comtab*** rather than ***standard-control-x-comtab*** since buffers are a feature of Zmacs, not of all Zwei-based editors.

16.6.2.5 Edit Buffers on m-X

```
    ZWEI:(SET-COMTAB *ZMACS-COMTAB*
                     ()
                     (MAKE-COMMAND-ALIST '(COM-EDIT-BUFFERS)))
```

This makes Edit Buffers available on m-X in Zmacs (by default it is only available on c-m-X).

16.6.2.6 m-. on m-(L)

```
    ZWEI:(SET-COMTAB *ZMACS-COMTAB*
                     '(#\m-MOUSE-L COM-EDIT-DEFINITION))
```

This makes clicking the left mouse button while holding down the META key do what m-. does. Invoking this command from the mouse is convenient when you specify the name of the definition to be edited by pointing at it rather than typing it.

16.6.3 Setting Mode Hooks

Each major mode has a *mode hook*, a variable which, if bound, is a function that is called with no arguments when that major mode is turned on.

16.6.3.1 Electric Shift Lock in Lisp Mode

```
(SETF ZWEI:LISP-MODE-HOOK 'ZWEI:ELECTRIC-SHIFT-LOCK-IF-APPROPRIATE)
```

This tells Lisp major mode to turn on Electric Shift Lock minor mode unless the buffer has a Lowercase attribute. The effect is that by default Lisp code is written in upper case.

16.6.3.2 Auto Fill in Text Mode

```
(SETF ZWEI:TEXT-MODE-HOOK 'ZWEI:AUTO-FILL-IF-APPROPRIATE)
```

This tells Text major mode to turn on Auto Fill minor mode unless the buffer has a Nofill attribute. The effect is that by default lines of text are automatically broken by carriage returns when they get too wide.

16.7 Customizing Zmail

The Profile command allows you to customize Zmail by setting various display and command options to your personal taste. You can set an option temporarily or permanently, the latter by saving the option in your *Zmail Profile.*

Classes of options you can set include the following:

- Format used for hardcopies of messages
- Mail-file attributes
- Lists of mail files and other objects that Zmail knows about at startup
- Associations between certain objects
- (M) actions for many top-level commands
- Screen configurations
- Default actions taken when reading, sending, replying to, or forwarding mail
- Command Tables

Customizing is done in *profile mode*, entered by clicking on [Profile] in the command menu at top level. The profile mode display (Figure 9) shows the text of your profile and the current settings of various options.

Setting and Saving Options

Option settings are stored in eight distinct places:

- *Your mind:* your conception of how the options should be set.
- *The Zmail environment:* the way the options are actually set at the moment.
- *The defaults:* the way the options are actually set before you alter them.
- *The editor buffer:* the in-memory buffer of your profile.
- *The source version of your profile:* on disk.
- *The compiled version of your profile:* also on disk.
- *Mail buffers:* options associated and stored with the individual mail buffers.
- *Mail files:* options associated with a mail buffer saved as a file.

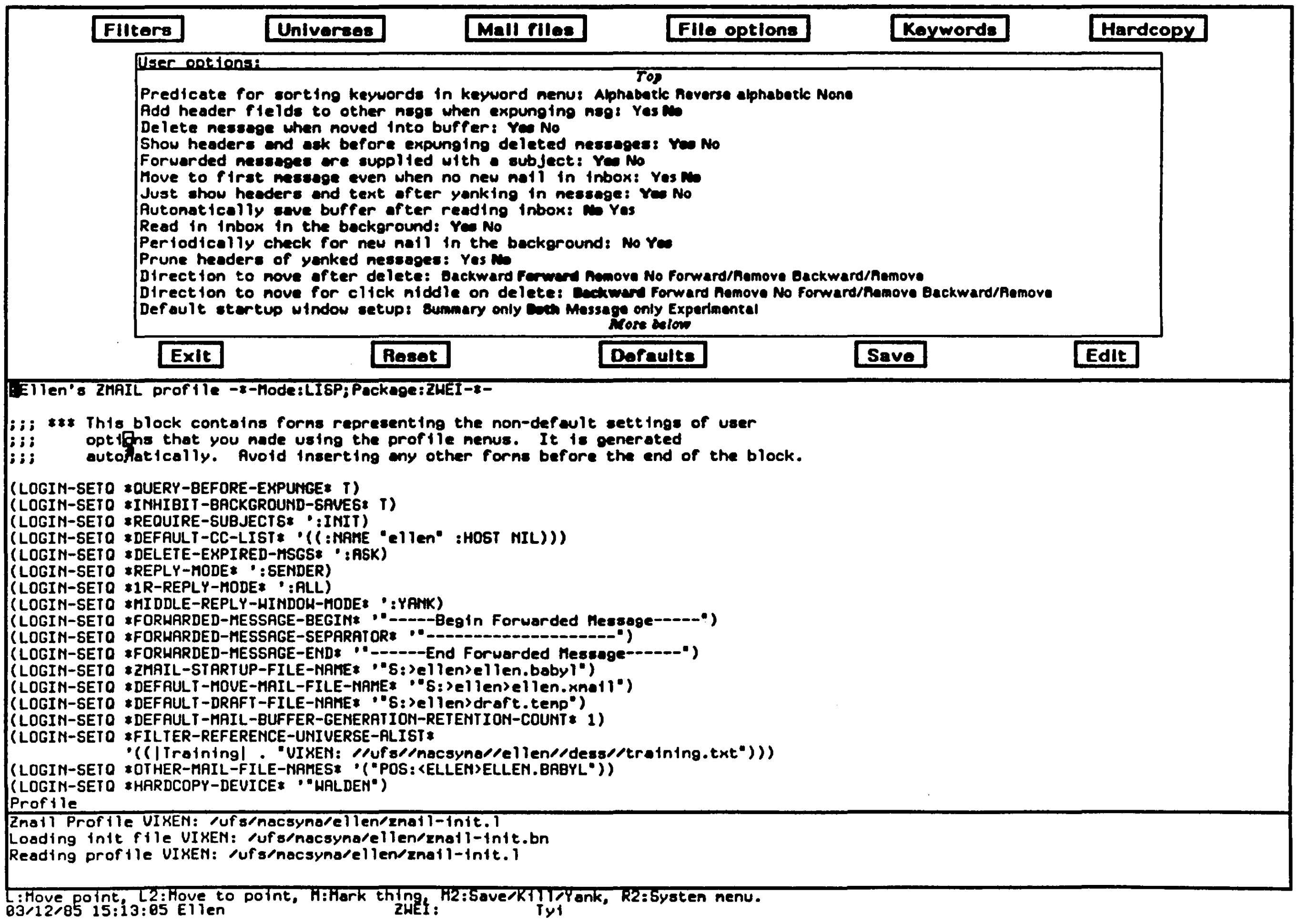
The simplest way to use profile mode is:

1. Make the changes you want using the menu items or user options window, two regions of the display indicated in Figure 9. For a list of the various options and what they mean: See the section "Zmail Profile Options" in *Communicating with Other Users*.

2. Use [Exit] to leave profile mode. Check to see that you like your changes.

3. To save your changes, reenter profile mode and use [Save]. Before you do this for the first time, use [Save (M)] and press RETURN to the question Zmail asks. This specifies that you want your file compiled, which makes it load and run faster. Answer *yes* to any questions about inserting changes or recompiling your file. At this point Lisp code corresponding to your option settings will be stored in your profile. Options changed using [File options] or [Keywords] are stored in the individual mail buffers and must be saved using [Save] on the top-level command menu.

What [Save] actually does is move option settings from the environment (where you altered them in the first step) to the editor buffer, then from the editor buffer to the source copy of your init file, and finally from the source file to the compiled file (by recompiling). You can also move option settings one step at time, by using [Reset] and [Default], and the menu options available by using [Save].



```
Filters    Universes    Mail files    File options    Keywords    Hardcopy

User options:
                                        Top
Predicate for sorting keywords in keyword menu: Alphabetic Reverse alphabetic None
Add header fields to other msgs when expunging msg: Yes [✓ No]
Delete message when moved into buffer: [✓ Yes] No
Show headers and ask before expunging deleted messages: [✓ Yes] No
Forwarded messages are supplied with a subject: [✓ Yes] No
Move to first message even when no new mail in inbox: Yes [✓ No]
Just show headers and text after yanking in message: [✓ Yes] No
Automatically save buffer after reading inbox: [✓ No] Yes
Read in inbox in the background: [✓ Yes] No
Periodically check for new mail in the background: No [✓ Yes]
Prune headers of yanked messages: Yes [✓ No]
Direction to move after delete: Backward [✓ Forward] Remove No Forward/Remove Backward/Remove
Direction to move for click middle on delete: [✓ Backward] Forward Remove No Forward/Remove Backward/Remove
Default startup window setup: Summary only [✓ Both] Message only Experimental
                                        More below

   Exit        Reset        Defaults        Save        Edit

Ellen's ZMAIL profile -*-Mode:LISP;Package:ZWEI-*-

;;; *** This block contains forms representing the non-default settings of user
;;;      options that you made using the profile menus.  It is generated
;;;      automatically.  Avoid inserting any other forms before the end of the block.

(LOGIN-SETQ *QUERY-BEFORE-EXPUNGE* T)
(LOGIN-SETQ *INHIBIT-BACKGROUND-SAVES* T)
(LOGIN-SETQ *REQUIRE-SUBJECTS* ':INIT)
(LOGIN-SETQ *DEFAULT-CC-LIST* '((:NAME "ellen" :HOST NIL)))
(LOGIN-SETQ *DELETE-EXPIRED-MSGS* ':ASK)
(LOGIN-SETQ *REPLY-MODE* ':SENDER)
(LOGIN-SETQ *1R-REPLY-MODE* ':ALL)
(LOGIN-SETQ *MIDDLE-REPLY-WINDOW-MODE* ':YANK)
(LOGIN-SETQ *FORWARDED-MESSAGE-BEGIN* '"-----Begin Forwarded Message-----")
(LOGIN-SETQ *FORWARDED-MESSAGE-SEPARATOR* '"--------------------")
(LOGIN-SETQ *FORWARDED-MESSAGE-END* '"------End Forwarded Message------")
(LOGIN-SETQ *ZMAIL-STARTUP-FILE-NAME* '"S:>ellen>ellen.babyl")
(LOGIN-SETQ *DEFAULT-MOVE-MAIL-FILE-NAME* '"S:>ellen>ellen.xmail")
(LOGIN-SETQ *DEFAULT-DRAFT-FILE-NAME* '"S:>ellen>draft.temp")
(LOGIN-SETQ *DEFAULT-MAIL-BUFFER-GENERATION-RETENTION-COUNT* 1)
(LOGIN-SETQ *FILTER-REFERENCE-UNIVERSE-ALIST*
            '((|Training| . "VIXEN: //ufs//macsyma//ellen//dess//training.txt")))
(LOGIN-SETQ *OTHER-MAIL-FILE-NAMES* '("POS:<ELLEN>ELLEN.BABYL"))
(LOGIN-SETQ *HARDCOPY-DEVICE* '"WALDEN")
Profile
Zmail Profile VIXEN: /ufs/macsyma/ellen/zmail-init.l
Loading init file VIXEN: /ufs/macsyma/ellen/zmail-init.bn
Reading profile VIXEN: /ufs/macsyma/ellen/zmail-init.l

L:Move point, L2:Move to point, M:Mark thing, M2:Save/Kill/Yank, R2:System menu.
03/12/85 15:13:05 Ellen                 ZWEI:           Tyi
```

Figure 9. Profile mode display

16.8 Customizing Converse

The following variables allow you to customize Converse's behavior. You can set them in your init file.

zwei:*converse-mode* *Variable*

Controls what happens when an interactive message arrives. It should have one of the following values:

:pop-up (This is the default.) A message window pops up at the top of the screen, displaying the message. You are asked to type R (for Reply), N (for Nothing), or C (for Converse). If you type R, you can type a reply to the message inside the message window. When you type END, this reply is sent back to whomever sent the original message to you, and the pop-up message window window disappears. If you type N, the message window disappears immediately. If you type C, the Converse window is selected. The input editor is used while you reply to a message in the pop-up message window, so you get some editing power, although not as much as with full Converse (since the latter uses Zwei). See the section "Using the Input Editor".

:auto The Converse window is selected. This is the window that shows you all of your conversations, letting you see everything that has happened, and letting you edit your replies with the full power of the Zwei editor. With this window selected, you can reply to the message that was sent, send new messages, participate in other conversations, or edit and write out messages or conversations. You can exit with c-END or ABORT (c-END sends a message and exits; ABORT just exits), or you can select a new window by any of the usual means (such as the FUNCTION or SELECT keys).

:notify A notification is printed, telling you that a message arrived and from whom. If you want to see the message, enter Converse by pressing SELECT C. There you can read the message and reply if you want to.

:notify-with-message
A notification is printed, which includes the entire contents of the message and the name of the sender. If you want to reply, you can enter Converse.

zwei:*converse-append-p* *Variable*

If the value is **nil** (the default), a new message is prepended to its conversation. If the value is not **nil**, a new message is appended to its conversation. **print-sends** is not affected by this variable; it always displays messages in forward chronological order.

zwei:*converse-beep-count* *Variable*

The value is the number of times to beep or flash the screen when a message arrives. The default value is two. Beeping or flashing occurs only if the Converse window is exposed or if the value of **zwei:*converse-mode*** is **:pop-up** or **:auto**. (Otherwise, notification tells you about the message and includes the usual beeping or flashing.)

zwei:*converse-end-exits* *Variable*

Controls the behavior of END and c-END. If ***converse-end-exits*** is set to **nil**, the default, END sends the message and you remain in Converse. c-END sends the message and exits Converse. Setting ***converse-end-exits*** to **t** reverses this, so that c-END sends the message and remains in Converse and END sends and exits.

16.9 Customizing Hardcopy Facilities

You can specify the printer you want to use for hardcopying files and screen images in your init file.

si:set-default-hardcopy-device *name* *Function*

si:set-default-hardcopy-device specifies the printer to be used for all of the hardcopy commands except the screen copy command. *name* is a string specifying the device name. This is the real name of the printer, its **name** attribute not its **pretty-name**. For example: `caspian-sea` is the real name of the printer whose pretty name is `Caspian Sea`. (The valid set of device names are the **printer** objects in your namespace database.)

si:set-screen-hardcopy-device *name* *Function*

si:set-screen-hardcopy-device specifies the printer to be used for screen copies (by the FUNCTION Q command). *name* is a string specifying the device name. This is the real name of the printer, its **name** attribute not its **pretty-name**. For example: `caspian-sea` is the real name of the printer whose pretty name is `Caspian Sea`. (The valid set of device names are the **printer** objects in your namespace database.)

You can specify personal default fonts for each device in your init file.

si:*hardcopy-default-fonts* *Variable*

si:*hardcopy-default-fonts* is a variable whose value is an association list where each element specifies a device and a set of keyword/value pairs designating the font. The keywords are **:default-font** and **:header-font**.

For example:

```
(login-forms
  (setq si:*hardcopy-default-fonts*
       '(("Itasca" :default-font "Fix18B"))))
```

in your init file will specify Fix18B as the default font for the printer Itasca.

You can use `Show Font HELP` in the Lisp Listener or the List Fonts (m-X) command in Zmacs to get a list of all the fonts that are currently loaded into the Lisp environment. The **fonts** package contains the names of all fonts. Here is a list of some of the useful fonts:

fonts:cptfont	This is the default font, used for almost everything.
fonts:jess14	This is the default font in menus. It is a variable-width rounded font, slightly larger and more attractive than medfnt.
fonts:cptfonti	This is a fixed-width italic font of the same width and shape as **fonts:cptfont**, the default screen font. It is most useful for italicizing running text along with **fonts:cptfont**.
fonts:cptfontcb	This is a fixed-width bold font of the same width and shape as **fonts:cptfont**, the default screen font.
fonts:medfnt	This is a fixed-width font with characters somewhat larger than those of **cptfont**.
fonts:medfnb	This is a bold version of **medfnt**. When you use Split Screen, for example, the [Do It] and [Abort] items are in this font.
fonts:hl12i	This is a variable-width italic font. It is useful for italic items in menus; Zmail uses it for this in several menus.
fonts:tr10i	This is a very small italic font. It is the one used by the Inspector to say *"More above"* and *"More below"*.
fonts:hl10	This is a very small font used for nonselected items in Choose Variable Values windows.
fonts:hl10b	This is a bold version of **hl10**, used for selected items in Choose Variable Values windows.

16.10 Changing the Default Printer

When a site has more than one printer, one of the printers is specified as the site default hardcopy device. You can change the default in your init file to specify the printer that is most convenient for you. See the function **si:set-default-hardcopy-device**, page 174.

In the System Menu, using [Hardcopy] allows you to specify a different printer name; the printer name is mouse-sensitive.

17. Checking on What the Machine is Doing

17.1 Poking Around in the Lisp World

This section describes a number of functions, most of which are not normally used in programs, but are "commands", that is, things that you type directly at Lisp. They are useful for finding information about your current state or about the Lisp world in general.

who-calls *symbol* &optional *pkg* (*do-inferiors* **t**) (*do-superiors* **t**) *Function*

symbol must be a symbol or a list of symbols. **who-calls** tries to find all the functions in the Lisp world that call *symbol* as a function, use *symbol* as a variable, or use *symbol* as a constant. (It does not find things that use constants that contain *symbol*, such as a list one of whose elements is *symbol*; it only finds it if *symbol* itself is used as a constant.) It tries to find all the functions by searching the function cells and properties of all the symbols in a certain set of packages. The set always includes the package *pkg*. If *do-inferiors* is true, the set also includes all packages that use *pkg*. If *do-superiors* is true, the set also includes all packages that *pkg* uses. *pkg* 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 *symbol* appears anywhere in the interpreted code. **who-calls** is smarter about compiled code.

If *symbol* is a list of symbols, **who-calls** does them all simultaneously.

The editor has a command, List Callers (m-X), that is similar to **who-calls**.

The symbol **unbound-function** is treated specially by **who-calls**. **(who-calls 'unbound-function)** searchs 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 whose names you misspelled 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.

The compiler records, as part of its debugging-info property, which top-level macros were expanded in the process of compiling it. This information is used by **who-calls** and similar functions. Thus you can use **who-calls** for macros. **who-calls** can also find callers of open-coded functions, such as substitutable functions. Functions compiled in earlier versions of the system have not recorded this information; hence **who-calls** will not be able to find them until those sources have been recompiled.

what-files-call *symbol-or-symbols* &optional *pkg* (*do-inferiors* **t**) (*do-superiors* **t**) *Function*

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 those files.

apropos *apropos-substring* &optional *pkg* (*do-packages-used-by* **t**) *do-packages-used* *Function*

Tries to find all symbols whose print-names contain *apropos-substring* as a substring. When 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. It checks all symbols in a certain set of packages. The set always includes *pkg*. If *do-packages-used-by* is true, the set also includes all packages that use *pkg*. If *do-packages-used* is true, the set also includes all packages that *pkg* uses. *pkg* defaults to the **global** package, so normally all packages are searched. **apropos** returns a list of all the symbols it finds. This is similar to the Find Symbol command, except that Find Symbol only searches the current package unless you specify otherwise.

where-is *pname* *Function*

Finds all symbols named *pname* and prints on **standard-output** a description of each symbol. The symbol's home package and name are printed. If the symbol is present in a different package than its home package (that is, it has been imported), that fact is printed. A list of the packages from which the symbol is accessible is printed, in alphabetical order. **where-is** searches all packages that exist, except for invisible packages.

If *pname* is a string it is converted to uppercase, since most symbols' names use uppercase letters. If *pname* is a symbol, its exact name is used.

where-is returns a list of the symbols it found.

The **find-all-symbols** function is the primitive that does what **where-is** does without printing anything.

describe *anything* &optional *no-complaints* *Function*

Tries to tell you all the interesting information about any object except array contents). **describe** knows about arrays, symbols, all types of numbers, packages, stack groups, closures, instances, structures, compiled functions, and locatives, and prints out the attributes of each in human-readable form. Sometimes it describes something that it finds inside something else; such recursive descriptions are indented appropriately. For instance, **describe** of a symbol tells you about the symbol's value, its definition, and each of its properties. **describe** of a floating-point number shows you its internal representation in a way that is useful for tracking down roundoff errors and the like.

If *anything* is a named-structure, **describe** handles it specially. To understand this: See the section "Named Structures" in *Reference Guide to Symbolics-Lisp*. 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 the symbolic names for the entries in the structure. **describe** knows how to use the names to print out each field's name and contents.

describe describes an instance by sending it the **:describe** message. The default method prints the names and values of the instance variables.

This is the same as the Show Object command.

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

inspect &optional *object* *Function*

A window-oriented version of **describe**. See the section "How the Inspector Works", page 189.

disassemble *function* *Function*

function is either a compiled function, or a symbol or function spec whose definition is a compiled function. **disassemble** prints out a human-readable version of the macroinstructions in *function*.

17.1.1 Variables for Examining the Lisp World

- *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 several values were printed (because of a multiple-value return), * is bound to the first value. If no result was printed, * is not changed.

// *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 **.

grindef *function-spec...* *Special Form*

Prints the definitions of one or more functions, with indentation to make the code readable. Certain other "pretty-printing" transformations are performed:

- The **quote** special form is represented with the ' character.
- Displacing macros are printed as the original code rather than the result of macro expansion.
- The code resulting from the backquote (`) reader macro is represented in terms of `.

The subforms to **grindef** are the function specs whose definitions are to be printed; ordinarily, **grindef** is used with a form such as **(grindef foo)** to print the definition of **foo**. When one of these subforms is a symbol, if the symbol has a value its value is prettily printed also. Definitions are printed as **defun** special forms, and values are printed as **setq** special forms.

If a function is compiled, **grindef** says so and tries to find its previous interpreted definition by looking on an associated property list. See the function **uncompile**. This works only if the function's interpreted definition was once in force; if the definition of the function was simply loaded from a BIN file, **grindef** does not find the interpreted definition and cannot do anything useful.

With no subforms, **grindef** assumes the same arguments as when it was last called.

break &optional *tag* (*conditional* **t**) *Special Form*

Enters a breakpoint loop, which is similar to a Lisp top-level loop. **(break** *tag***)** always enters the loop; **(break** *tag conditional***)** evaluates *conditional* 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.
```

The standard values for any variables are checked. If **break** rebinds any of these standard variables, it warns you that it has done so. **break** 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 pressed, control is returned to the previous break or Debugger, or to top level if there is none. If the RESUME key is pressed, **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 the variable **sys:*break-bindings***, page 181.)

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

There are two easy ways to write a breakpoint into your program: **(break)** gets a read-eval-print loop, and **(dbg)** gets the Debugger. (These are the programmatic equivalents of the SUSPEND and m-SUSPEND keys on the keyboard.)

sys:*break-bindings* *Variable*

When **break** is called, it binds some special variables under control of the list that 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. You can **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**.

dbg:*debugger-bindings* *Variable*

When the Debugger is entered, it binds some special variables under control of the list that is the value of **dbg:*debugger-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. You can **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 the Debugger.

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**. See the section "Initializations" in *Internals, Processes, and Storage Management*.

17.2 Utility Functions

zwei:save-all-files &optional (*ask* **t**) *Function*

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 command Save File Buffers (m-X) does, but it stays away from redisplay and other advanced facilities so that it might work if other things are broken.

zwei:zmail-save-all-files is similar, but saves mail files from Zmail.

print-sends &optional (*stream* **standard-output**) *Function*

Prints out all messages you have received (but not messages you have sent), in forward chronological order, to *stream*. Converse is more useful for looking at your messages, but this function predates Converse and is retained for compatibility.

print-notifications &optional (*from* **0**) (*to* **(1- (length tv:notification-history))**) *Function*

Reprints any notifications that have been received. The difference between notifications and sends is that sends come from other users, while notifications are asynchronous messages from the Lisp Machine system itself. If *from* or *to* is specified, prints only part of the notifications list.

Example: `(print-notifications 0 4)` prints the five most recent notifications.

This is the same as the Show Notifications command.

si:print-login-history &optional (*whole-history* **si:login-history**) *Function*

Prints one line for each time the **login** function has been called in this world load. Each line contains the name of the user that logged in, the name of the machine on which the world load was running at that time, and the date and time. If you cold boot, log in, and then call **si:print-login-history**, the last line refers to your own login and all previous lines refer to logins that were done before running **disk-save**.

This information is useful to determine how many times a world load has been disk-saved, on what machines it was disk-saved, and who disk-saved it.

The first couple of lines do not contain any date or time, because they were made during the initial construction of the world load before it found out the current time. Names of users at other sites that are not in the local site's namespace search list are qualified with the site's namespace name and a vertical bar. The user SCRC|LISP-MACHINE is the dummy user used by **si:login-to-sys-host** at SCRC, the site where new world loads are created.

hostat &rest *hosts* *Function*

Asks each of the *hosts* for its status, and prints the results. If no hosts are specified, all hosts on the Chaosnet are asked. Hosts can be specified by either name or octal number.

For each host, a line is displayed that either says that the host is not responding or gives metering information for the host's network attachments. If a host is not responding, probably it is down or there is no such host at that address. A Lisp Machine can fail to respond if it is looping inside **without-interrupts** or paging extremely heavily, such that it is simply unable to respond within a reasonable amount of time.

To abort the host status report produced by **hostat** or FUNCTION H, press c-ABORT.

uptime &rest *hosts* *Function*

Queries the specified *hosts*, asking them for their "uptime"; each host responds by saying how long it has been up and running. **uptime** prints out the results. If **uptime** reports that a host is "not responding", either the host is not responding to the network, or it does not support the UPTIME protocol.

The **uptime** function is a variant of **hostat**.

17.3 Dribble Files

Sometimes it is useful to have a more permanent record of what is happening on your screen when a program is running. Dribble files allow you to save the output from or interaction with a program in a file or a buffer. Formerly such files were called wallpaper files because the resulting long strips of paper output resembled wallpaper and were sometimes posted on the wall to demonstrate the operation of a program. Now that display consoles are in wide use, the files are referred to as dribble files because the output "dribbles" out of the running program.

dribble-start *pathname* &optional *editor-p* (*concatenate-p* **t**) *Function*

Opens *filename* as a "dribble file". It rebinds **standard-input** and **standard-output** so that all of the terminal interaction is directed to the file as well as to the terminal. If *editor-p* is non-**nil**, then instead of opening *filename* on the file computer, **dribble-start** directs the terminal interaction into a Zmacs buffer whose name is *filename*, creating it if it does not exist.

dribble-end *Function*

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

17.4 status and sstatus

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

status *status-function* &optional (*item* **nil** *item-p*) *Special Form*

(status features) returns a list of symbols indicating features of the Lisp environment. The default list for the Lisp Machine is:

```
(:DEFSTORAGE :LOOP :DEFSTRUCT :LISPM :SYMBOLICS 3600 :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. The order of this list should not be depended on, and might not be the same as shown above.

The following symbols in the features list can be used to distinguish different Lisp implementations, using the #+ and #- reader syntax.

Three symbols indicate which Lisp Machine hardware is running:

:lispm	Any kind of Lisp Machine, as opposed to Maclisp
:cadr	An M.I.T. CADR
:3600	A 3600-family machine

One symbol indicates which kind of Lisp Machine software is running:

:symbolics	Symbolics software

See the section "Sharp-sign Reader Macros" in *Reference Guide to Symbolics-Lisp.*

(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, for example, "**MIT-LISPM-6**". Note that this is not the same as the value of **site-name**.

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

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

sstatus *status-function item* *Special Form*

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

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

17.5 Using Peek

17.5.1 Peek

You start up Peek by pressing SELECT P, by using the Select Activity Peek command, or by evaluating **(peek)**.

The Peek program gives a dynamic display of various kinds of system status. When you start up Peek, a menu is displayed at the top, with one item for each system-status mode. The item for the currently selected mode is highlighted in reverse video. If you click on one of the items with the mouse, Peek switches to that mode. Pressing one of the keyboard keys as listed in the Help message also switches Peek to the mode associated with that key. The Help message is a Peek mode; Peek starts out in this mode.

Pressing the HELP key displays the Help message.

The Q command exits Peek and returns you to the window from which Peek was invoked.

Most of the modes are dynamic: they update some part of the displayed status periodically. The time interval between updates can be set using the Z command. Pressing *n*Z, where *n* is some number, sets the time interval between updates to *n* seconds. Using the Z command does not otherwise affect the mode that is running.

Some of the items displayed in the modes are mouse sensitive. These items, and the operations that can be performed by clicking the mouse on them, vary from mode to mode. Often clicking the mouse on an item gives you a menu of things to do to that object.

The Peek window has scrolling capabilities, for use when the status display is longer than the available display area. See the section "Scrolling".

As long as the Peek window is exposed, it continues to update its display. Thus a Peek window can be used to examine things being done in other windows in real time.

17.5.2 Peek Modes

Processes (P)

In Processes mode, invoked by pressing P or by clicking on the [Processes] menu item, you see all the processes running in your environment, one line for each. The process names are mouse sensitive; clicking on one of them pops up a menu of operations that can be performed:

Arrest (or Un-Arrest)	Arrest causes the process to stop immediately. Unarrest causes it to pick up where it left off and continue.
Flush	Causes the process to go into the state Wait Forever. This is one way to stop a runaway process that is monopolizing your machine and not responding to any other commands. A process that has been flushed can be looked at with the debugger or inspector and can be reset.
Reset	Causes the process to start over in its initialized state. This is one way to get out of stuck states when other commands do not work.
Kill	Causes the process to go away completely.
Debugger	Enters the Debugger to look at the process.
Describe	Displays information about the process.
Inspect	Enters the Inspector to look at the process.

See the section "Introduction: Processes" in *Internals, Processes, and Storage Management.*

Areas (A)

Areas mode, invoked by pressing A or by clicking on [Areas], shows you information about your machine's memory. The first line is hardware information: the amount of physical memory on the machine, the amount of swapping space remaining in virtual memory, and how many wired pages of memory the machine has. The following lines show all the areas in virtual memory, one line for each. For each area you are shown how many regions it contains, what percentage of it is free, and the number of words (of the total) in use. Clicking on an area inserts detailed information about each region: its number, its starting address, its length, how many words are used, its type, and its GC status. See the section "Areas" in *Internals, Processes, and Storage Management.*

Meters (M)

Meters mode, invoked by pressing M or by clicking on [Meters], shows you a list of all the metering variables for storage, the garbage collector, and the disk. There are two types of storage and disk meters:

Timers	Timers have names that start with ***ms-time-** and keep a total of the milleseconds spent in some activity.
Counts	Counts have names that start with ***count-** and keep a running total of the number of times some event has occurred.

The garbage collector meters fall into two groups according to which part of the garbage collector they pertain to: the scavenger or the transporter. See the section "Operation of the Garbage Collector" in *Internals, Processes, and Storage Management.*

File System (F)

File System mode, invoked by pressing F or by clicking on [File System], provides you information about your network connections for file operations. For each host the access path, protocol, user-id, host or server unit number, and connection state are listed. For active connections information about the actual packet flow is also given. The various items are mouse sensitive. For hosts, you can get hostat information, do a file reset, log in remotely, find out who is on the remote machine, and send a message to the machine. You can reset, describe, or inspect data channels, and close streams.

Resetting an access path makes the server on a foreign host go away, which might be useful to free resources on that host or if you suspect that the server is not working correctly.

Windows (W)

Windows mode, invoked by pressing W or clicking on [Windows], shows you all the active windows in your environment with the panes they contain. This allows you to see the hierarchical structure of your environment. The items are mouse sensitive. Clicking on a window name pops up a menu of operations that you can perform on the window.

Servers (S)

Clicking on [Servers] or pressing S puts Peek in Servers mode. If your machine is a server (for example, a file server), Servers mode shows the status of each active server.

Network (N)

Network mode, invoked by pressing N or by clicking on [Network], shows information about the networks connected to your machine. For each network there are three headings for information:

Active connections	The data channels that your machine has opened to another machine or machines on the network.
Meters	Information about the data flow (packets) between your machine and other machines on the network.
Routing table	A list of all the subnets and for each the route to take to send packets to a host on that subnet.

To view the information under one of these headings, you click on the heading. The hosts and data channels in the list of active connections are mouse sensitive. For hosts, you can get hostat information, do a file reset, login remotely, find out who is on the remote machine, and send a message to the machine. You can reset, describe, or inspect data channels.

Information about the hardware network interface is also displayed, as well as metering variables for the networks.

Hostat (H)

Clicking on [Hostat] or pressing H starts polling all the machines connected to the local network. For each host on the network a line of information is displayed. Those machines that do not respond to the poll are marked as "Host not responding". You terminate the display by pressing c-ABORT.

Help and Quit

Clicking on the [Help] menu item displays the help information that is displayed when Peek is selected the first time.

Clicking on [Quit] buries the Peek window and returns you to the window from which you invoked Peek.

18. Tools for Lisp Debugging

18.1 How the Inspector Works

The Inspector is a window-oriented program for inspecting data structures. When you ask to inspect a particular object, its components are displayed. The particular components depend on the type of object; for example, the components of a list are its elements, and those of a symbol are its value binding, function definition, and property list.

The component objects displayed on the screen by the Inspector are mouse-sensitive, allowing you to do something to that object, such as inspect it, modify it, or give it as the argument to a function. Choose these operations from the menu pane at the top-right part of the screen.

When you click on a component object itself, that component object gets inspected. It expands to fill the window and its components are shown. In this way, you can explore a complex data structure, looking into the relationships between objects and the values of their components.

The Inspector can be part of another program or it can be used standalone; for example, the Display Debugger can utilize some of the panes of the Inspector. Note, however, that although the display looks the same as that of the standalone Inspector, the handling of the mouse buttons depends upon the particular program being run.

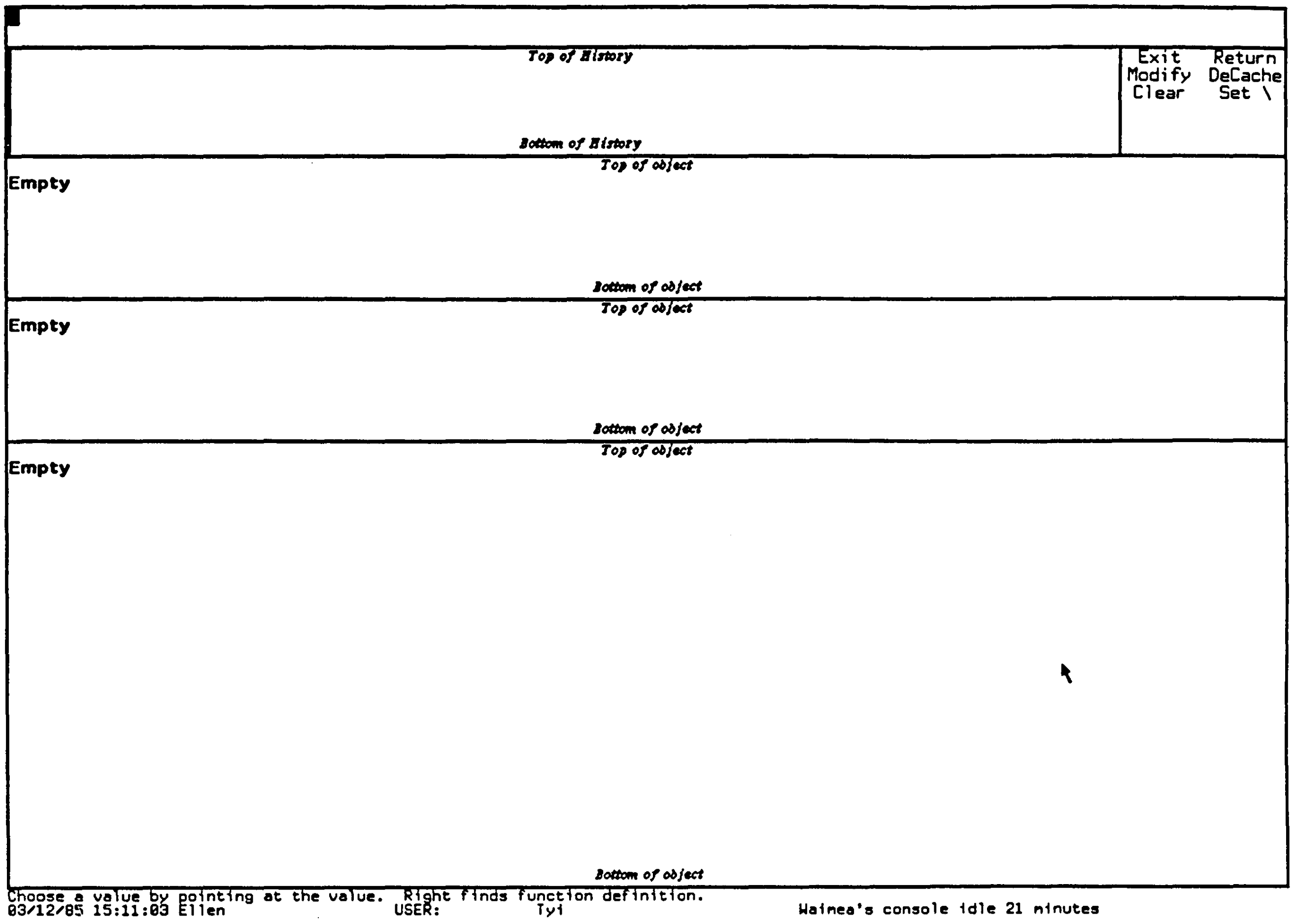
Figure 10 shows the standalone Inspector window. The display consists of the following panes, from top to bottom:

- A small interaction pane
- A history pane and menu pane
- Some number of inspection panes (three by default)

18.2 Entering and Leaving the Inspector

You can enter the standalone Inspector via:

- Select Activity *Inspector*
- SELECT I
- [Inspect] in the System menu
- The Inspect command, which inspects its argument, if any



```
Choose a value by pointing at the value.  Right finds function definition.
03/12/85 15:11:03 Ellen          USER:       Tyi               Wainea's console idle 21 minutes
```

Figure 10. The Inspector

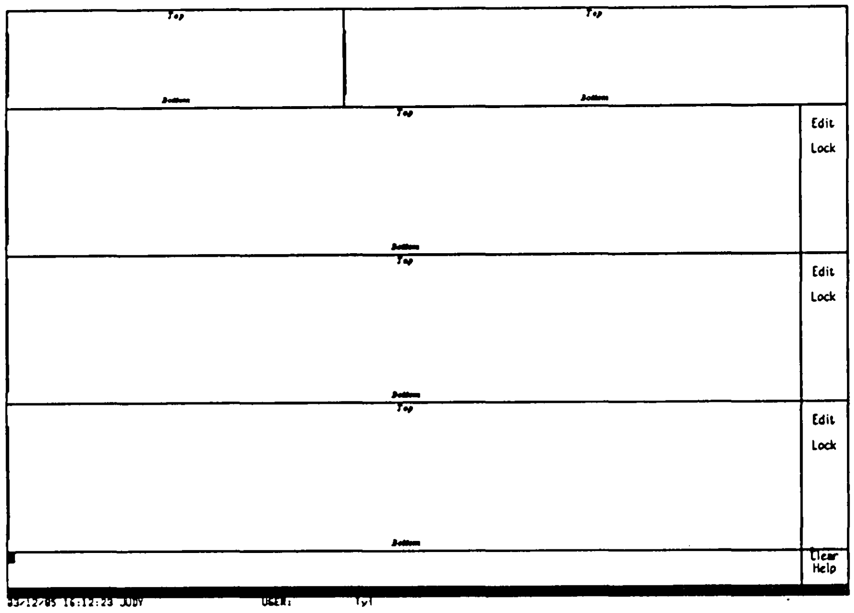
- The **inspect** function, which inspects its argument, if any

Warning: If you enter with the Inspect command or the **inspect** function, the Inspector is not a separate activity from the Lisp Listener in which you invoke it. In this case you cannot use SELECT L to return to the Lisp Listener; you should *always* exit via the [Exit] or [Return] option in the Inspector menu. If you forget and exit the Inspector by selecting another activity, you might need to use c-m-ABORT to return the Lisp Listener to its normal state.

18.3 Flavor Examiner

The Flavor Examiner utility examines the structure of flavors defined in the Lisp environment. You can select the Flavor Examiner with either SELECT X or the System menu.

The Flavor Examiner window is divided into six panes.



The examiner panes (the three middle panes) list the answer to a query. The edit item of each examiner pane places the contents of the pane into a Zmacs possibilities buffer. The lock item for a examiner pane prevents the pane from being updated.

You enter a flavor name or method-spec into the interaction pane (the bottom pane).

To get started, type the name of a flavor in the interaction pane.

Methods are listed in the following format:

```
MESSAGE-NAME method-type method-combination-type FLAVOR
```

If the method-combination-type is **:case**, this format is used:

```
MESSAGE-NAME SUBMESSAGE-NAME method-type method-combination-type FLAVOR
```

Clicking on a flavor results in these actions:

- A left click on a flavor presents a menu of flavors and methods related to the flavor. (Note that automatically generated methods to get and set instance variables and methods associated with **si:vanilla-flavor** are not listed.)
- A middle click on a flavor presents a menu of related instance variables.
- A right click on a flavor presents a menu of operations on the flavor, including edit and inspect.
- Any click places on a flavor it in the flavor history pane if it is not already there.

Clicking on a method results in these actions:

- A left click on a method lists the instance variables to which the method refers.
- A middle click on a combined method lists the methods used to build the combined method.
- A middle click on a noncombined method lists all methods for that message from any flavor.
- A right click on a method presents a menu of operations on the method, including [arglist], [documentation], [edit], [inspect], [method spec], [trace], and [disassemble], unless the method is pseudocombined.
- Any click on a method places it in the method history if it is not already there.

Clicking on an instance variable results in these actions:

- A left click on an instance variable lists the methods that refer to the instance variable.
- A middle click on an instance variable shows the default value of the instance variable.

18.4 Entering the Debugger

When an error condition is signalled and no handlers decide to handle the error, an interactive Debugger is entered to allow you to look around and see what went wrong and to help you continue the program or abort it. This section describes how to use the Debugger and the various debugging facilities.

18.4.1 Entering the Debugger by Causing an Error

The Debugger is invoked automatically when errors arise during program execution or when you explicitly cause an error, for example, by typing a nonsense symbol name, such as **ahsdgf**, at the Lisp read-eval-print loop.

18.4.1.1 Error Display

Errors are signalled by the microcode and by Lisp programs (by using **ferror** or related functions). Here is an example of an error:

```
foo

>>Trap: The variable FOO is unbound.

SI:*EVAL:
   Arg 0 (FORM): FOO

s-A, RESUME: Supply a value to use this time as the vaue of FOO
s-B, m-C:    Supply a value to store permanently as the value of FOO
s-C:         Retry the SYMEVAL instruction
s-D, ABORT:  Return to Lisp Top Level in Lisp Listener 1
→
```

>> indicates entry to the Debugger. The word immediately following >> shows what caused you to enter the Debugger; most commonly you see `Trap`, `Error`, or `Break`.

`Trap` indicates a microcode error.
`Error` indicates a software error.
`Break` indicates entry by keystroke or the **dbg** function.

The message that follows describes the error in English, in this example, an unbound variable. The next two lines in the example show the stack frame in which the error occurred — the function that was being called and the current value(s) of its argument(s).

The right-facing arrow (→) indicates that the Debugger is waiting for a command. Multiple arrow prompts signal recursive invocations of the Debugger.

Debugger Proceed and Restart Options

The Debugger provides options for proceeding from the error or restarting from some prior point. When the Debugger is entered, all *proceed types*, *special commands*, or *restart handlers* available in the error context are assigned to keystrokes with the SUPER modifier, starting with s-A, s-B, and so on, from the most recently established (innermost) to the oldest (outermost). Also, the RESUME key is assigned to the innermost proceed type (or restart handler if there are no proceed types), and the ABORT key is assigned to the innermost restart handler. All these keystroke assignments are displayed when you enter the Debugger or when you type the c-L Debugger command. (See the section "Conditions" in *Reference Guide to Symbolics-Lisp*.)

You can use one of these options or any of the Debugger commands. See the section "How to Use the Debugger" in *Program Development Utilities*. For details on the Debugger command keys: See the section "Special Keys" in *Reference Guide to Symbolics-Lisp*.

Optionally, you can request that backtrace information appear when you enter the Debugger by setting the variable **dbg:*show-backtrace*** in your init file.

18.4.2 Entering the Debugger with m-SUSPEND

You can also enter the Debugger explicitly by pressing m-SUSPEND. Adding the CONTROL modifier to this combination has the effect of saying "enter the Debugger immediately". Thus, you can:

- Press m-SUSPEND while the currently running program or read-eval-print loop is reading from the console.
- Press c-m-SUSPEND so that the currently running program enters the Debugger whether or not it is reading from the console.

Note: Pressing the SUSPEND key without the META modifier or just pressing c-SUSPEND enters a read-eval-print loop rather than the Debugger.

18.4.3 Entering the Debugger with the dbg Function

You can use the **dbg** function in your source code to help detect errors in your programs.

- Insert a call to **dbg** (with no arguments) into your code and then recompile.
- Call **dbg** with an argument of *process* to force a process into the Debugger.

dbg &optional *process* *Function*

Forces *process* into the Debugger so that you can look at its current state. **dbg** sets up a restart handler for c-Z, ABORT, and RESUME that exits from the **dbg** function back to the original process. The message for this restart handler is "Allow process to continue". You can use c-T, c-R, c-m-R, and other similar Debugger commands when you enter the Debugger via **dbg**.

- With no argument, it enters the Debugger as if an error had occurred for the current process. It is not an error; in particular, **errset** and **catch-error** do not handle it. You can include this form in program source code as a means of entering the Debugger. This is useful for breakpoints and causes a special compiler warning.
- With an argument of **t** (rather than a process, window, or stack group), it finds a process that has sent an error notification.

Suppose you are running in process *X* and you use **dbg** on some process *Y*. Process *Y* is forced into the Debugger, no matter what it is doing.

Technically, it is "interrupted", similar to how c-SUSPEND, c-ABORT and c-m-SUSPEND work. Process *Y* starts running the Debugger, using the stream **debug-io**. **debug-io** gets the same stream as was bound to **terminal-io** in Process *X*. At this time, Process *X* waits in a state called DBG until Process *Y* leaves the Debugger, and so Process *X* does not contend for the stream.

For more information: See the special form **break**, page 180. See the section "Breakpoints" in *Reference Guide to Symbolics-Lisp*.

Appendix A
Front-end Processor

A.1 Introduction to the FEP

Symbolics computers use a front-end processor known as the FEP. The FEP is a small computer inside the processor cabinet, based on a microcomputer chip. It plays several roles in the operation of the system, the most visible being booting (loading and starting the software of) the Lisp system.

This discussion corresponds to FEP software version 17 or higher, which are required for Release 6.0. Use the Show Version command to determine the FEP version of your machine. See the section "Commonly Used FEP Commands", page 200. If you have a FEP of version lower than 17, contact Field Service.

A.2 Hints on Using the FEP

The Fep> prompt is displayed when you are at FEP command processor level. The FEP command processor provides defaults and documentation where appropriate. When using it, remember these hints:

- You need type only enough of a FEP command to identify it uniquely, as shown below:

Input	*Completes to*
b RETURN	Boot
l w RETURN	Load World >World1.load
st RETURN	Start

- You can press the HELP key for a list of all FEP commands. For example:

```
HELP

Possible choices are:  Add Boot Clear Continue
Copy Disk Dismount Halt Load Mount Reset
return-keyboard-to-lisp set show start test
```

Some of these commands are used in ordinary booting; others exist primarily for system maintainers, to help them debug unusual problems.

- You can press the HELP key after typing a command name for a list of all possible completions to that command. For example:

  ```
  set HELP

  Possible choices are:  chaos-address
  default-disk-unit disk-type
  microcode-name-and-version
  ```

  Note that you must press SPACE after typing a command name and before pressing HELP to receive a list of the command's arguments.

- You can insert parenthetical comments in any white space within or after FEP commands. For example:

  ```
  load world >World1.load (contains geological survey programs)
  set chaos-address 401 (ACME-earthquake)
  ```

  `load world >World1.load` and `set chaos-address 401` are FEP commands, and the parenthetical phrases are user-supplied comments. Such comments make useful documentation for configuration files.

- Finally, be careful! If you make a mistake when giving FEP commands, you might leave the machine in a state from which it cannot be cold booted.

A.3 Cold Booting

Cold booting completely resets Lisp. When you are finished using the computer, you can cold boot it to put it into a fresh state for the next user. Avoid cold booting a machine that someone else may be using, though, since the other person may be expecting the machine to remain in its current state.

First, log out if you are logged in. Then, halt the machine by typing the Halt Machine command to a Lisp Listener. (The function **sys:halt** can also be used.)

If you cannot obtain a Lisp Listener window, or if no Lisp Listener is responding to keyboard input, press h-c-*upper-left*. However, **sys:halt** is the preferred way to stop Lisp than h-c-*upper-left* because h-c-*upper-left* might interrupt disk I/O operations.

With FEP version 18 and later versions, pressing h-c-*upper-left* does not immediately stop Lisp. Instead, the FEP asks Lisp to stop itself cleanly. If Lisp does stop itself, the FEP prints the message "Lisp stopped itself." If Lisp does not stop itself after about three seconds, the FEP prints, "Waiting for Lisp to stop itself..." If after another three seconds Lisp does not stop itself, the FEP forcibly stops Lisp and prints, "Halting execution of Lisp." The purpose of this behavior is to reduce the chance of halting the computer during a disk write, which might cause ECC errors.

When control has returned to the FEP, type the following FEP command to cold boot the machine:

```
boot file-name RETURN
```

where *file-name* is a configuration file. Its default value is the last file name given the Boot or Show File command. Its initial default value is >Boot.boot on the current default disk unit. The following is a typical configuration file:

```
clear machine
load microcode >Microcode1.mic
load world >World1.load
set chaos-address 401
start
```

Alternatively, if the microcode is already loaded and the Chaosnet address is set, you can type these FEP commands manually:

```
load world >World1.load
start
```

File names specified in a command file refer by default to the disk unit containing the command file. For example, the FEP command Load World >World1.load, if contained in the file fep0:>Boot.boot, loads the world file fep0:>World1.load.

Cold booting takes approximately one minute. It takes another minute for Lisp to start. During this time, the machine might print a message asking you to enter date and time information if it has no other way to find it. (This behavior is site dependent.) If so, type it in the following format:

```
09/21/83 15:04
```

Be sure to enter the date and time correctly, as it is important that the file system know exactly when files are created and modified. If the calendar clock has been set, the machine uses the calendar clock reading as the default time for you to type in. If the calendar clock has not been set, it offers to set it to the time you specify.

A.4 Resetting the FEP

Resetting the FEP restarts the FEP system, thereby discarding knowledge of the FEP's free storage area. Resetting might be necessary if you unplug the console video cable from either end or turn the console off and on. You also need to reset the FEP if you receive the error message: `Request for` *N* `longs failed`. You can reset the FEP from either the keyboard or the processor front panel.

- To reset the FEP from the keyboard:
  1. Type the form `(si:halt)` to stop the computer and give control of the keyboard to the FEP.

2. Type the command Reset Fep to the FEP prompt.

Alternatively, if no Lisp Listener is responsive:

1. Press h-c-*upper-left* to stop the computer and give control of the keyboard to the FEP.
2. Type the command Reset Fep.

Press y to answer the confirmation prompt.

- To reset the FEP from the processor front panel:
  1. Push the red RESET button on the processor front panel.
  2. Press the spring-loaded YES switch to answer the "Reset FEP?" question.

After you reset the FEP, the keyboard is connected to the FEP, not to Lisp. Give the Start command and press RETURN to warm boot the machine and Lisp, and return control of the keyboard to Lisp.

A.5 FEP Commands

Some FEP commands are involved in normal use of the computer. These include Boot, Show Directory, Show Version, and Start. Other commands are used primarily by system maintainers to debug unusual problems. Among these are Clear Machine, Disk Restore, Disk Format, and Load World. Be careful when giving these latter commands. If you make a mistake, you might destroy the state of the loaded or saved Lisp system.

A.5.1 Commonly Used FEP Commands

Boot *file-name* Boot executes the commands specified in *file-name*. *file-name* is the name of a configuration file; it defaults to the last file name given the Boot or Show File command. Its initial default is >Boot.boot.

Show *subcommand*

Show has several subcommands:

Show Configuration

Displays the hardware configuration, scans the backplane, and describes the boards that exist on the Lbus.

Show Directory *directory-spec*

Displays the contents of a directory in the FEP file system and the associated file comments. *directory-spec* must end in >* and can be preceded by fep: or fep*n*:. For example, Show Directory >* is acceptable. The default *directory-spec* is >*.

Use Show Directory to see whether the FEP is able to access the disk properly.

Show Disk-label *unit*	Displays the label of *unit*, the specified disk unit. This is done independently of the unit's being mounted, so you can tell what the label seems to contain. The default for *unit* is the current default disk unit.
Show File *file-name*	Displays the contents of *file-name*, a file in the FEP file system. *file-name* defaults to the last file name given the Show File or Boot command. Its initial default is >Boot.boot.
Show Status	Displays the internal status of some machine registers. For information on interpreting the output of this command: See the section "Finding Out Why Your Machine Crashed", page 214. See the section "FEP Show Status Command Output", page 217.
Show Version	Displays the version number of loaded FEP software.

A.5.2 Less Common FEP Commands

Add Disk-type

Lets you declare an arbitrary disk type to the FEP. You can declare up to four disk types before you have to give the Clear Disk-types command. Add Disk-Type is needed only to format and restore disks. It is not needed for normal operation of any validly formatted disk with a FEP file system.

Add Disk-type has the following arguments, for which it prompts with the argument names in parentheses:

name	The textual name by which this disk type is known
cylinders	The number of cylinders supported by the drive
heads	The number of heads on the drive
sectors	The number of sectors
gapl	The length of "gapl"

gap2	The length of "gap2"
gap3	The length of "gap3"
fast	0 for slower disks, 1 for faster disks

These numbers require careful computation and involve some restrictions of the computer hardware. The calculation should be done by Symbolics personnel.

Add Paging-file *file-name*

Adds the data pages of *file-name* to the paging tables. *file-name* defaults to >page.page on the currently selected disk unit. This command is executed implicitly during a Load World command. Beginning with V23 proms, >reserve.page is the default for Add Paging-file. For earlier proms, >page.page is the default.

Note: If you are executing a Boot command, Add Paging-file uses the same disk unit as the command file to find the page file. If you type in Load World, Add Paging-file uses the default disk unit, which might be different from that containing the world file just loaded, to find the page file.

If you are typing in this command or if you are adding more than one paging file, you should first use the Clear Paging-files command. Neither the FEP nor Lisp has any error checking to detect doubly allocated disk blocks. Not clearing the paging files will eventually cause surprising errors to occur after booting.

The Start command notifies you if no paging file has been established. It prompts for confirmation before starting the processor.

Clear *subcommand*

This command has the following subcommands:

Clear Disk-types	Clears all disk types declared with the Add Disk-type command.
Clear Machine	Clears the internal state of the registers and memories.
Clear Screen	Clears the console's screen.
Clear Paging-files	Clears all information in memory about what part of the disk holds virtual memory. Use this command after loading a world and before using an Add Paging-file command and starting the processor.

Continue

Continues the computer's operation from where it left off. However, if you have stopped the world and loaded new microcode, Continue does not work. Instead, you must warm boot by using Start.

Copy — Not currently implemented.

Disk Format — Formats the disk. This command overwrites all data on the disk. When you give the command, the FEP asks several questions; it expects answers in the following form:

Questions	*Valid answers*
Of type	M2284, T306, M2351, or D2257
On unit	Disk unit number
With pack id	0
From cylinder	Cylinder number; inclusive lower bound
Through cylinder	Cylinder number; inclusive upper bound

Disk Restore — Restores the file system or files from cartridge tapes to disk. Note: Before using Disk Restore, first ensure that memory is clear and the appropriate microcode is in place, by giving the commands Clear Machine and Load Microcode Tape:. Note the trailing colon (:) after Tape.

Using Disk Restore causes three questions to be displayed:

1. Have you used Set Disk-type for all units that do not have valid label blocks?

   The disk type must be known before the first reference to it, in case the label block is not yet written.

2. Is there a microcode at the beginning that should be skipped?

   The program that generates tapes to be used for disk restoring writes microcode followed by an EOF (end-of-file) mark so the FEP can easily skip it, since the microcode is in a stream format and the tape restoration data is not. Usually, continuation tapes (used, for example, when a world is too big to fit on a single reel of tape) do not begin with microcode; only the first reel begins with microcode.

   A tape can contain information in either *image restore* (*block restore*) format or *file restore* format. In both cases, up to 1152 characters of information are displayed, describing the contents of the section of the tape. Usually the information is supplied by the person producing the tape.

- *image restore*: Data recorded in this format can be either an initial file system or raw disk blocks from a source disk. The tape-generating program writes out block numbers normalized to block 0 (and writes the number of the original starting block into the header information) so that they can be written to the new disk at a different location if desired. File systems *must* be restored to block 0; the header information reminds you of this.

- *file restore*: Data recorded in this format must be a file. The header includes the name of the source file, its length, and a comment supplied by the writer of the tape. You are asked for a destination pathname for the data; the default disk unit is assumed unless another is specified. Both a file system and the specified destination file must already exist on the unit, and the destination file must be large enough to hold the tape data. If the data pages of the destination file are not contiguous (because of bad blocks, say, or lack of contiguous space), then the restored data is fragmented also.

  When the file restoration is completed, a special restoration block is read, containing the length of the file, the author, the creation date, and a comment.

  Large files (for example, worlds not compacted by garbage collection) cross tape boundaries. But since all block numbers are relative to the beginning of the file, the second reel of tape is logically continuous with the first, and file restoration proceeds as for single-reel files.

3. Do you want to restore it?

   Answering "no" skips the current tape restore section and searches for the next one.

Dismount *disk-unit*

Dismounts the unit, forcing a remount on the next reference. The default value of *disk-unit* is the current default disk unit.

Halt

Halts the FEP. To restart (and reset) it, push the RESET button on the processor front panel. The Halt command is being replaced by the Shutdown command.

Load Fep *file-name*

Loads and starts loadable FEP programs. The names of the FEP programs are usually of the form Vxx-name, where xx is the number of the FEP version on which the program runs and name is the name of the program. For example, the file name FEP:>V22-debug.flod would indicate that the program ran on FEP version 22 and was used for debugging Lisp crashes.

Load Microcode *file-name*
: Loads microcode memory and other high-speed memories from the specified file. The default value of *file-name* is the last file name given the Load Microcode command. Its initial default is >Microcode1.mic. Use Clear Machine first if the computer was just powered on.

Load Sync-program *file-name*
: Loads the specified file (of type .SYNC) into the sync program memory of the I/O board and causes the screen to clear. This is used for machines with monitors that require different sync programs than the one that is preprogrammed into the FEP. The default value of *file-name* is the last file name given the Load Sync-program command. Its initial default is >Sync.sync.

Load World *file-name*
: Restores enough of the saved world in the computer so that you can start up the machine. It prints both the desired microcode version for this world and the currently loaded microcode version. The default value of *file-name* is the last file name given the Load World command. Its initial default is >World1.load.

Mount *disk-unit*
: Reads the disk label of the specified disk unit to learn the number of cylinders, number of heads, fast mode, pack id, and location of the root directory. The FEP does an implicit Mount whenever files are referenced and the unit has not previously been mounted. The default value of *disk-unit* is the current default disk unit.

Reset *subcommand*
: Resets various parts of the computer. Subcommands include:

Reset Cart	Resets the cartridge tape drive.
Reset Clock	Resets the processor clock.
Reset Disk *unit-number*	Selects, fault-clears, and recalibrates the unit, clearing any error conditions. The default value of *unit-number* is the current default disk unit.
Reset Fep	Restarts the FEP program, discarding the FEP's knowledge of what microcode was loaded, and so on. If the FEP is running in RAM, asks whether to switch back to PROM.

Reset Lbus	Resets the Lbus.
Reset Most	Resets the processor clock, the Lbus, the sequencer, the video, and the disks. If you think that the internal state of the computer is inconsistent, try Reset Most before power-cycling.
Reset Sequencer	Resets the sequencer data paths.
Reset Video	Reloads the console screen's sync program.

Return-keyboard-to-lisp

Returns control of the keyboard to Lisp from the FEP. This command is used if the computer has been started via debugging cables instead of by typing to the FEP. It tells the FEP to send keyboard input to Lisp rather than process it as FEP commands.

Set *subcommand* This command has the following subcommands:

Set Chaos-address *octal-value*

Sets the Chaosnet address. The default value of *octal-value* is the previous Chaosnet address, which is set to zero when the FEP is started.

The FEP checks for an acceptable Chaosnet address before starting Lisp. If none is specified as argument to this command, it warns you, asks whether the current setting is acceptable, and allows you to change it if necessary.

Set Default-disk-unit *unit*

Sets the default disk unit. *unit* becomes the default for most subsequent disk references. However, within a command file executed by a Boot command, the default disk unit is the one on which the command file is located.

Set Disk-type *unit type pack-id*

Tells the FEP that disk *unit* is of type *type* and has pack id *pack-id*. Disk Restore might need this information if the disk has no label block or if the label block contains incorrect information. Give Set Disk-type after an implicit or explicit Mount command.

Set Display-string *string*

Displays the string in the nanofep display of machines that have a nanofep display. The length of the string is limited to 12 characters, the number of characters in the nanofep display. If more characters are used, the string is truncated. This command can be used in a .boot file.

Set Microcode-name-and-version *name version*
Sets *name* and *version* in the computer's memory after the world is loaded. Use this command if the FEP warns that it does not know that microcode is loaded. The information is also recorded for subsequent Load World commands.

If you load microcode, restart the FEP, and load a world, the FEP warns that the microcode has not been loaded and that it therefore cannot set its name and version in the computer's memory. Set Microcode-name-and-version is chiefly for use by Symbolics developers.

Set Monitor-type *monitor-type*
Specifies the monitor type. The Set Monitor-type commmand ensures that the sync program used is for the monitor type requested. *monitor-type* can be either **Moniterm** or **Philips**; the types can be abbreviated to their first letter, **m** for **Moniterm** and **p** for **Philips**.

Set Monitor-type is used if the monitor is changed at a site and the ID prom is not changed accordingly. This command can be used in a boot file.

The following examples show two valid uses of the same command.

```
Set Monitor-type Moniterm

set mon m
```

Shutdown

Halts the FEP. To restart (and reset) it, push the RESET button on the processor front panel. The preferred way to turn off the machine is:

- Halt the 3600-family processor, using **si:halt**.

- Halt the FEP, using the Shutdown command.

- Power off the processor and console.

On the 3600, the Shutdown command asks "Do you really want to halt the FEP?" and displays the message "FEP Halted" in the nanofep display.

On the 3670, the Shutdown command asks the question "Do you really want to power down the 3600?" and it lights the fault light on the switch panel.

Note: the Shutdown command replaces the Halt command.

Start	Starts the computer. If the world has just been loaded, this is a cold boot; if the world had been loaded previously, this is a warm boot. Start checks for an acceptable network address; if none was set, it is read from the computer and you are asked to confirm it.
Test	Performs simple tests of main memory, A memory, and the disks. Currently, the Test command is chiefly for use by Symbolics developers. A full set of Test commands is scheduled for a future release.

A.6 FEP File System Overview

The Symbolics computer disk has a file system called the *FEP file system*. The entire disk is divided up into *FEP files* (that is, files of the FEP file system). FEP files have names syntactically similar to those of files in the Symbolics computer's own local file system. However, the FEP file system and the Lisp Machine File System (LMFS) are completely distinct.

FEP files (for example, fep0:>Boot.boot) can be accessed from Lisp. The following files are part of the FEP file system and should never be disturbed.

```
>disk-label.fep
>root-directory.dir
>free-pages.fep
>bad-blocks.fep
>sequence-number.fep
```

A.6.1 Microcode Loads

By convention, files of type MIC are microcode loads. These files contain images of the microcode and the contents of other internal high-speed memories that are initialized when the computer is booted.

A.6.2 World Loads

By convention, FEP files of type LOAD contain world loads, or *bands* (images of entire Lisp worlds).

A.6.3 Configuration Files

Configuration files contain FEP commands tailored for a particular Lisp Machine configuration. The commands are executed if you specify the file as argument to a Boot command when cold booting the machine. See the section "FEP Commands", page 200.

The configuration file >Boot.boot usually contains FEP commands to:

- Clear the internal state of the machine
- Load the microcode
- Load a world
- Set the Chaosnet address
- Start the machine

To change the selection of microcode and world loads that are booted by default, simply use Zmacs to edit the file >Boot.boot. Be careful to avoid typographical errors; otherwise, you might have to type in the commands manually in order to boot the machine. Also, be sure that the last command in the file is followed by RETURN.

A.6.4 How LMFS Uses the FEP File System

The FEP file fep0:>lmfs.file is where LMFS normally keeps its files. It holds the machine's local file system. The entire Symbolics computer local file system normally resides inside one big file of the FEP file system.

The file fep0:>fspt.fspt tells LMFS which FEP files to use for file space, if not fep0:>lmfs.file.

A.6.5 Virtual Memory

The FEP file fep0:>page.page holds the virtual memory of the Lisp system while Lisp is running. To increase the effective size of virtual memory, you can add additional paging files. See the section "Allocating Extra Paging Space" in *Reference Guide to Streams, Files, and I/O*.

A.6.6 FEP File Comment Properties

Comment properties supply additional information about the contents of FEP files. In the Dired mode of Zmacs, they are listed inside square brackets, where the reference or expunge date appears for other file systems. You can list the contents of the FEP file system by using the function **print-disk-label**. The Zmacs command Dired (m-X) of fep:>*, or the form `(dired "fep:>*")` invokes the directory editor on the FEP file system. An example of the Zmacs Dired command output follows:

```
48150 free, 322330/370480 used (87%)
  FEP0:>BAD-BLOCKS.FEP.1  776      0(8)    9/14/83 11:46:56 [List of bad blocks] rll
  FEP0:>boot.boot.15      1    121(8)      1/15/84 12:19:15 [] DEG
  FEP0:>boot.boot.16      1    121(8)      1/29/84 13:06:43 [] DEG
  FEP0:>boot.boot.17      1    121(8)      2/21/84 13:35:28 [] whit
  FEP0:>boot.boot.18      1    124(8)      2/21/84 13:39:20 [] whit
  FEP0:>DISK-LABEL.FEP.1   24      0(8)    9/14/83 11:46:55 [The disk label] rll
  FEP0:>FREE-PAGES.FEP.1   41      0(8)    9/14/83 11:46:56 [Free pages map] rll
  FEP0:>fspt.fspt.1     1      0(8)        9/14/83 11:46:58 [A filesystem partition table] rll
  FEP0:>LMFS.file.1 50000     0(8)         1/05/84 23:20:13 [] ptaylor
  FEP0:>Microcode1.MIC.1  103 117020(8)    6/30/83 08:19:16 [TMC5-MIC 219] Feinberg
  FEP0:>PAGE.PAGE.1 150000     0(8)        9/14/83 11:46:58 [Main paging area] rll
  FEP0:>Release-5-0.load.1 19109 22013568(8) 11/02/83 17:02:31 [Release 5 Beta Test] joseph
  FEP0:>ROOT-DIRECTORY.DIR.1     2      0(8) 9/14/83 11:46:55 [His highness] rll
  FEP0:>sequence-number.fep.1     1      0(8) 9/14/83 11:49:39 [] rll
  FEP0:>System-243-463.load.1 22348 25733376(8) 1/02/84 11:46:14 [Exp 243.463] Zippy
  FEP0:>system-243-481.load.1 20544 23666688(8) 1/11/84 22:48:56 [Exp 243.481, Full-GC]
  FEP0:>system-243-516.load.1 20754 23908608(8) 1/24/84 23:23:41 [Exp 243.516, Full-GC] Zippy
  FEP0:>system-243-559.load.1 19157 22068864(8) 2/19/84 19:32:45 [Exp 243.559, Full-GC] Moon
  FEP0:>TMC5-MIC.MIC.247  103 118018(8)    10/03/83 20:25:07 [TMC5-MIC 247, Beta Test] joseph
  FEP0:>TMC5-MIC.MIC.262  101 115233(8)    12/27/83 21:15:16 [TMC5-MIC 262] whit
  FEP0:>TMC5-MIC.MIC.273  101 115810(8)    2/19/84 15:13:56 [TMC5-MIC 273] whit
  FEP0:>World1.load.1 19138 20318976(8)    10/07/83 12:09:08 [Rel 4.5] LISPMNIL
```

A.6.7 Installing Microcode

Use **si:install-microcode** to retrieve any new microcode from the file system of the sys host.

si:install-microcode *from-file-or-version* &optional *to-file-or-version* *boot-file-to-update* *Function*

Installs microcode from a system file into a file in the FEP file system.

from-file-or-version is a microcode version number (in decimal). The file resides in the logical directory sys:l-ucode;.

to-file-or-version rarely needs to be supplied. It defaults to a file on FEP:> (the root directory of the boot disk) whose name is based on the microcode name and version. If supplied, *to-file-or-version* is either a pathname (string) of a file on FEP:>, or an integer *n*, which stands for the file TMC5-MIC.MIC.n on FEP:>.

The logical directory *sys: l-ucode;* includes multiple types of microcode for each version number. The correct microcode to install depends upon the particular hardware configuration of your machine. When your machine is shipped, the default microcode filename is correct, but if your machine is upgraded (for example, an FPA board is installed) you might need to override the default used by **si:install-microcode** to get the correct type for your configuration. Below is an example of how you would get the microcode for a 3600 running 6.0, with no console upgrade but an FPA board installed:

```
(si:install-microcode "tmc5-fpa-mic.mic.319")
```

The correct microcode types for each system and hardware configuration are shown below. The names in this table omit the suffix mic.*n* that you must include to indicate the version of the required microcode. The version number must be followed by a period. Microcode version 319. is required for Release 6.0.

3600

	tmc5	**ifu**
No FPA	tmc5-mic.	ifu-mic.
FPA	tmc5-fpa-mic.	ifu-fpa-mic.

3670/3600 with console upgrade

	tmc5	**ifu**
No FPA	tmc5-io4-mic.	ifu-io4-mic.
FPA	tmc5-io4-fpa-mic.	ifu-io4-fpa-mic.

3640

	tmc5	**ifu**
No FPA	tmc5-io4-st506-mic.	ifu-io4-st506-mic.
FPA	tmc5-io4-st506-fpa-mic.	ifu-io4-st506-fpa-mic.

If you use the wrong microcode for your configuration, your machine will not boot, except in the case where your system has an FPA and you use a non-FPA microcode. In this case, the machine functions normally, but does not make use of the FPA at all.

boot-file-to-update specifies whether to update the boot file with the new microcode version number. It accepts one of these values (the default value is **nil**):

Value	*Action*
nil	Prompts for a boot file to update.
pathname	Does not prompt but uses pathname as the boot file to update.

:no-boot-file-update	Does not prompt or update.

Initially, the Symbolics personnel who install your system establish these microcode files for you.

A.6.8 Renaming FEP Files

FEP files can be renamed. For example, if you save a world containing MACSYMA, you might want to rename the world file to >macsyma.load or >macsyma1.load. Be sure to update your boot file if you intend this to be the default world.

A.6.9 Using a Spare World Load for Paging

You can reuse FEP file space by specifying a FEP file, usually a spare world load file, to be used as an extension of the file >page.page. To do so, use the FEP Add Paging-file command. Note that this action overwrites the previous contents of the specified file.

You should rename this file to, say, >extra.page, so that other users do not attempt to use the file space (to hold customized world loads, for instance).

You can also create new FEP files and use them for extra paging space. See the section "Allocating Extra Paging Space" in *Reference Guide to Streams, Files, and I/O*.

A.6.10 Adding a Spare World Load as LMFS File Space

Partitions can be added to LMFS with [Local FS Maint] on the File System Editor menu. Select this item to get a menu of file-system maintenance operations. The [Initialize (R)] command yields a menu of initialization options, which offers [New File System] and [Auxiliary Partition] as a choice. [New File System] is similar to [Initialize (L)]; it initializes a partition to be the basis of a file system.

When you add a new partition or a partition on another disk, the disk should be free of errors and properly initialized and formatted, and the partition should exist.

To add another partition, use [Auxiliary Partition]. Enter the pathname of the FEP file to be used as the new partition. The default presented, which is correct for [New File System], is never correct for adding a partition. Then use [Do It]. The system then performs much verification and error checking, roughly as much as when initializing a new partition. It must not be interrupted while performing these actions. When finished, it adds the partition and edits the FSPT automatically.

A.7 Disk Handling

You can include a disk specification of the form fep*n*:, where *n* refers to disk unit *n*, as the first field of file and directory references to the FEP. A specification of fep: (with no unit number) refers to the disk unit from which the current Lisp world was booted, that is, the unit containing the world load file. If fep*n*: is omitted entirely, the default disk unit, set by Set Default-disk-unit is assumed. See the section "Less Common FEP Commands", page 201.

A.7.1 Disk Handling Commands

The following FEP commands manipulate disk units. See the section "Commonly Used FEP Commands", page 200. See the section "Less Common FEP Commands", page 201.

- Add Disk-type
- Clear Disk-types
- Dismount *disk-unit*
- Mount *disk-unit*
- Reset Disk *unit-number*
- Set Default-disk-unit *unit*
- Set Disk-type *unit type pack-id*
- Show Directory *directory-spec*
- Show Disk-label *unit*
- Show File *file-name*

A.7.2 Multiple Disk Units

Each Lisp Machine can access more than one local disk drive. The following conditions apply:

- The FEP can access any drive at all. Currently, the hardware allows a maximum of four drives.
- You can boot a Lisp world from any drive by using the FEP command Load World. Also, you can add paging files from any drive by using the FEP command Add Paging-file.

- The form fep: refers to disk 0; it is equivalent to the form fep0:. However, it is also possible to specify another disk explicitly, using such forms as fepl: or fep2:.

- You can access only drive 0 from Lisp. In particular, you cannot **disk-save** to any drive other than drive 0. This means that users cannot save customized world loads onto any other drive, or store user files on any other drive.

- World loads are not specific to a type of drive. This means that if one Lisp Machine has a T306 drive and another has an M2284 drive, world loads can be transferred back and forth between the drives.

A.7.2.1 Disk Types

The FEP currently supports the following types of disk drive:

- M2351 (470 megabytes unformatted capacity)

- T306 (300 megabytes unformatted capacity — removable)

- M2284 (167 megabytes unformatted capacity)

- D2257 (167 megabytes unformatted capacity)

- Maxtor XT-1140 (140 megabytes unformatted capacity)

A.8 Finding Out Why Your Machine Crashed

When your machine crashes, the FEP Show Status command displays information that can be useful in diagnosing the cause of the crash. For an outline of the information that Show Status prints: See the section "FEP Show Status Command Output", page 217.

The Show Status output section "3600 program counters" includes the macro PC, the CPC, and the 16 OPCs. The macro PC is the address of the current instruction of compiled Lisp code. The CPC is the address of the current microinstruction. The OPCs are the addresses of the 16 most recently executed microinstructions; OPC+0 is the most recent, OPC+17 the earliest. An arrow points at either the CPC or the first OPC, depending on the error condition that stopped the machine. This is the microinstruction that was executing when the event occurred that was the proximate cause of the machine's stopping itself.

A.8.1 Decoding Micro PCs

Use the function **dbg:decode-micro-pc** to decode the microcode PCs printed by the FEP command Show Status.

dbg:decode-micro-pc *pc* &optional (*name* **sys:%microcode-version**) (*version* **(sys:microcode-version-number sys:%microcode-version)**) *Function*

dbg:decode-micro-pc is useful for investigating why a machine crashed. It decodes the octal microinstruction addresses printed by the FEP command Show Status. To use this function you should first write down the Show Status output. You can then either warm boot the machine using the Start command or call **dbg:decode-micro-pc** on another machine.

pc is an address in the microcode, taken from the CPC or OPC information printed by the Show Status command. Show Status prints these numbers in octal; if your default radix is decimal, precede *pc* by **#o**. Normally the number in the Show Status output with the arrow (→) pointing to it is the relevant number, but it can sometimes be useful to try decoding all of the numbers to get additional clues.

name and *version* are optional; they specify the version of the microcode that was running at the time of the crash. You can omit these arguments if you call **dbg:decode-micro-pc** while using the machine that crashed and while running the same microcode version as at the time of the crash. You can also omit these arguments if you call this function from another machine that has a software *and* hardware configuration that is *identical* to that of the machine that crashed. To find the microcode version name and number that a machine is running, use **(print-herald :verbose t)** or take the name and version number of the microcode file in the machine's boot file (normally fep0:>Boot.boot). Microcode version numbers are decimal; include a period at the end of the number if your default radix is octal.

Example:

```
(dbg:decode-micro-pc #o44552 "tmc5-mic" 253.)
```

dbg:decode-micro-pc prints information that depends on the microinstruction:

Microinstruction	*Information printed*
Halt instruction	The reason it halts the machine. An example is "error in the error handler". These reasons are constant strings in the microcode source program and do not represent any dynamic analysis of the state of the machine.
Signaller of a Lisp error	The internal form of the error message.

This is not the same form of error message you would ever see otherwise; normally Lisp software translates these messages into conditions and signals them, and the conditions define more readable error messages. This is useful mainly in decoding OPCs earlier than the one with the arrow, when the machine halted because of "error in the error handler".

Handler for a macroinstruction in compiled Lisp code
The name of that macroinstruction. A halt here might be caused by running a world together with an incompatible microcode, such as a microcode from an earlier release, that does not implement an instruction used by that world.

If all else fails, the function offers to load the microcode symbol table (from the sys:l-ucode; directory) and then prints the symbolic name of the microinstruction. Loading the microcode symbol table takes a few minutes. Microinstruction symbolic names can sometimes be clues to help in figuring out what the machine was doing at the time it crashed.

Two types of symbolic names exist: those with and without parentheses.

If the name includes parentheses, it is a list of the name of a microcode routine and the path through that routine to reach the microinstruction in question. Beware of a pitfall! These names are not unique; the same microinstruction can be reached by multiple paths from different microcode routines. For example, a microinstruction named (FTN-AR-1 3) might also be part of the microcode for the CAR instruction; you cannot assume too much from the name if it contains parentheses. It is only a clue.

If a symbolic name is just a symbol and has no parentheses, it is unique and names the first microinstruction of a microcode routine.

Beware of assuming too much. If the reason Lisp stopped itself is not "microcode halted", the information that **dbg:decode-micro-pc** prints is not likely to be helpful, though it might be useful to people who understand the hardware.

A.8.2 Decoding Macro PCs

To decode the macrocode PC printed by the FEP command Show Status, warm boot or go to another machine running identical software and call the function **%find-structure-header** on the number printed by the FEP. This is an octal number; use **#o** if necessary. It should return a compiled-function object, which is the function that was executing at the time. To find the exact place in the function that was executing, note the different between the number printed by the FEP and the address in the printed representation of the compiled-function object. You can use **%pointer-difference** to compute this difference. Multiply this by 2, and add 1 if the FEP said the PC was odd (not even). The result is the instruction number of the current instruction; disassemble the compiled function to see it.

Example:

```
Fep>Show status
...
3600 program counters:
  Macro PC/ (Odd)1244531
  ...
Fep>Start
...
(%find-structure-header #o1244531)
#<DTP-COMPILED-FUNCTION EQUAL 1244530>
(%pointer-difference #o1244531 *)
1
(1+ (* * 2))
3
(disassemble ***)
  0 ENTRY: 2 REQUIRED, 0 OPTIONAL
  1 PUSH-LOCAL FP|0          ;A
  2 PUSH-LOCAL FP|1          ;B
  3 BUILTIN EQL STACK
  ...
```

Instruction 3 (EQL) is the one that halted.

A.9 FEP Show Status Command Output

The register contents and program counters displayed by the Show Status command give some information on machine states causing the FEP message "Lisp stopped itself". They are generally not useful for interpreting wired-ferror halts. Show Status merely prints the contents of certain hardware registers, decoding the bits symbolically. The FEP does not interpret these contents, so some output might not be meaningful. The following cautions apply:

- You must interpret some bits depending on the value of other bits.
- Some registers listed below are printed only if they contain "useful" information.

The most important registers are *Sequencer status* and *MC error status*. For information on decoding the PCs printed elsewhere: See the section "Finding Out Why Your Machine Crashed", page 214.

FEP buffer status

Bit	*Meaning*
Spy DMA Enb	Spy bus being used by FEP to access disk or net (means spy bus being used for normal functions)
Write to dev / Read from dev	Spy DMA direction
Drive busy	Spy DMA mode (who controls busy line)
Int Enb	Spy DMA enable to interrupt FEP
Count up / Count down	Spy DMA address increment direction
Busy	SPY DMA busy (inside FEP)
Spy DMA busy	Spy DMA busy line (actual line on backplane)
DMA setup	[meaning unknown]

FEP Lbus control

Bit	*Meaning*
ECC Diag	Normal memory error correction logic disabled; instead, FEP can read or write the 8 extra bits of main memory
Doorbell Int Enb	Doorbell (Lisp-to-FEP signal) interrupt enabled
Use Uncorrected Data	FEP unaware of corrected Lbus data if single-bit-error
Ignore Double ECC Error	FEP does not get bus error if uncorrectable Lbus error (either double-bit error or nonexistent memory)
Task 3 Req	FEP trying to wake up microtask 3
Doorbell	Doorbell ringing (Lisp-to-FEP signal)
Lbus Buffer Busy	[self-explanatory]
Lbus Buffer Some Parity Error	[self-explanatory]

FEP Board ID control

Bit	*Meaning*
Continuity	Read-back of random signal that checks board presence
Lbus ID Req	Lbus reading board IDs, not doing normal functions
Half Speed	Main processor clock running at half speed

FEP Proc control

Bit	*Meaning*
Lbus Power Reset	Reset all Lbus devices due to power turn-on or turn-off
Lbus Reset	Reset all Lbus devices
Clear Errors	Bit that clears FEP error registers (not an error)
FEP Int Enable	FEP interrupt enable (not an error)
Kept Alive	FEP died and was reset by nanofep
Lbus Power Reset (on bus)	Same as above, but actually read back from the bus
Lbus Reset (on bus)	[...]
FEP Ram Par Err	Parity error in dynamic ram on FEP board

Sequencer error status

(Status of the SQ board and the main error status bits that can halt the machine)

Bit	*Meaning*
Microcode-halted	A "halt" microinstruction was executed, for one of the following reasons: • A call to the %HALT function (due to a wired-ferror or a call to HALT) • A fatal error, such as an error while entering the error handler or an error in wired code (page fault, disk handlers) • Executing an undefined macroinstruction (running too old a version of microcode or executing bad macrocode) • Failure of a microcode consistency check (stack frame too large, stack overwritten)

Self-explanatory hardware errors:

Bit	*Meaning*
Spare-error-bit	[never happens, unless manually wired to some signal]
GC-Map-parity-error	GC MAP ram on DP board
Type-map-parity-error	TYPE MAP ram on DP board
Page-Tag-parity-error	PAGE TAG ram on FEP board
A-memory-parity-error	AMEM ram on DP board
B-memory-parity-error	BMEM ram on DP board
MC-error (map, ifu, or main mem)	Error on MC board; see MC error status

AU-error	Error on AU (FPA) board (if the machine has one)
Task-state-memory-parity-error	TSKM ram on SQ board (doesn't always halt machine)
Control-memory-parity-error	CMEM ram on SQ board (also for microcode breakpoints if an L-Console program is cabled up for debugging)

Hardware "errors" that are not always errors:

Bit	*Meaning*
CTOS-low-parity-error	CSTK ram on SQ board (low half of output register)
CTOS-high-parity-error	CSTK ram on SQ board (high half of output register)

(Note: If CTOS-came-from-IFU (see below) is true, above two bits have no meaning.)

Sequencer miscellaneous status

(Status bits that are not errors)

Bit	*Meaning*
CTOS-came-from-IFU	CTOS register holds macroinstruction dispatch address from IFU (or TMC) rather than contents of CSTK ram
TSK-STOP (sequencer stopped)	Machine is stopped for some reason
Errhalt-Sync	Some error bit is on (stops machine)
MC Wait	Microinstruction waiting for memory control to allow it
Task Switch	Switching to a different microtask

MC Error status

Bit	*Meaning*
Double bit error	An uncorrectable error in main memory, or a reference to a non-existent Lbus address. Further information in ECC syndrome (see below).
Map A parity error, Map B parity error	Parity errors in the map caches on the MC (TMC, IFU) board.
Hit in both map A and map B	Both map caches claiming to map the same address. Could be the map hardware, or some hardware or microcode problem causing map to be written with bad data.

ECC syndrome

(An octal number followed by an address with x's in it)

This register contains the most recent main-memory read-error correction status. The error can be caused by a read by the processor, a read by the FEP, or a read by a DMA I/O device. The events that set this register include nonexistent memory reference, single-bit error correction, and double-bit error detection. Non-existent memory and double-bit error halt the processor (even if it was the FEP or an I/O device that got the error). Currently, the FEP disables itself from getting a bus error if it references nonexistent Lbus memory or gets a double-bit error in Lbus memory.

One other event that can set this register is a bug in the FEP's code for examining the machine's status. Generally, the first two digits of the address are 77 in this case.

The address is the physical address of the location referenced. Only bits 23-18 and 1-0 are valid (the rest are x'ed out). These are sufficient bits to determine which Lbus slot (bits 23-19) and which of the 8 banks within a memory board are being referenced. To convert the address to an Lbus slot number, consider the one or two digits at the left of the x's to be an octal value, and divide it by 2. This is a logical slot number, as printed (in decimal) by the Show Configuration command. It is not related to the numbers printed on the machine chassis. Slot 0 is at the left, as seen from the front of the machine.

The syndrome codes are as follows:

	0	1	2	3	4	5	6	7
000	okay	36	37	2-bit	38	2-bit	2-bit	3
010	39	2-bit	2-bit	6	2-bit	8	9	2-bit
020	40	2-bit	2-bit	13	2-bit	15	16	2-bit
030	2-bit	18	19	2-bit	20	2-bit	2-bit	unused
040	41	2-bit	2-bit	23	2-bit	25	26	2-bit
050	2-bit	28	29	2-bit	30	2-bit	2-bit	31
060	2-bit	33	34	2-bit	35	2-bit	2-bit	unused
070	NXM	2-bit	2-bit	unused	2-bit	unused	unused	2-bit
100	42	2-bit	2-bit	0	2-bit	1	2	2-bit
110	2-bit	4	5	2-bit	7	2-bit	2-bit	10
120	2-bit	11	12	2-bit	14	2-bit	2-bit	unused
130	17	2-bit	2-bit	unused	2-bit	unused	unused	2-bit
140	2-bit	21	22	2-bit	24	2-bit	2-bit	unused
150	27	2-bit	2-bit	unused	2-bit	unused	unused	2-bit
160	32	2-bit	2-bit	unused	2-bit	unused	unused	2-bit
170	2-bit	unused	unused	2-bit	unused	2-bit	2-bit	unused

3600 program counters

Label	*Meaning*
Macro PC	The address of the current instruction of compiled Lisp code. This is prefaced with either (Odd) or (Even) since there are two instructions per word.
Current micro PC (CPC)	The address of the current microinstruction.
Old PCs (OPC)	The addresses of the 16 most recently executed microinstructions. OPC+0 was executed most recently, OPC+17 least recently.

A.10 Debugging in the FEP

The release tapes include some files provided as an extra debugging aid. These files can be used to enter a debugging mode in the FEP. This mode is especially useful for problems that cause control to return to the FEP, making it impossible to use the debugging methods normally used in Lisp.

These files have names of the form: v*n*-debug.flod, where *n* is the FEP version number. The files should now reside on your sys host in the directory with the logical pathname sys: l-fep; v*nn*-debug flod where *nn* indicates the version of FEP software.

To use these files, you should copy the appropriate file to the FEP file system *before* you need to use it. To copy the file, first find out which version of FEP software is installed in your machine. You can do this by either typing the Lisp function **print-herald** to Lisp or by typing the Show Version command at the FEP level. To copy this file to your FEP file system, use the Zmacs command Copy File (m-X).

For example, if you are using FEP version 22 software, you would use the following command to copy the .flod file to the FEP file system:

```
Copy File (m-X) sys:l-fep;v22-debug.flod fep0:>v22-debug.flod
```

The FLOD file cannot be used on any other FEP version; trying to use one on a different FEP version has no effect.

After you have copied the file to the FEP file system, you can enter the debugging mode by loading the file with the Load Fep command, as shown in the following example:

```
Fep>Load Fep >v22-debug.flod
```

This puts you into a debugging mode very similar to the Debugger under Lisp, whereby you can move up and down the stack to examine the state of the machine and determine the source of the problem. The HELP key lists the commands that are available.

One particularly useful command, when the machine has crashed during paging, is c-m-S. This command allows you to switch between the auxiliary stack (where paging code runs) and the normal stack (where user code runs). If the machine crashed while executing on the auxiliary stack, user stack frames will not be found until c-m-S is executed.

sys:halt &optional *fep-commands* *Function*

The function **si:halt** stops execution of Lisp and gives control to the FEP. This function does not interrupt disk I/O operations when it stops Lisp.

The function **sys:%halt** should no longer be used as it can halt disk operations. Interrupting a disk write can cause a fatal ECC error later, because the contents of the disk block are incomplete. This can render directories and other files inaccessible and is a particular problem when halting the machine while using LMFS.

If the optional *fep-commands* are not supplied or if the argument is **nil**, **si:halt** places you at the FEP level. If you supply a string for *fep-commands*, Lisp causes the FEP to interpret the string as if the string were typed in from the keyboard or read from a command file. It is important to use a new line for each command that is part of the string.

The following form halts Lisp, passes control to the FEP and causes the Show Version and Continue commands to be executed.

```
(si:halt "Show Version
Continue
")
```

This example generates the following display:

```
(si:halt "Show Version
Continue
")
Lisp stopped itself.
Closing command file.
Fep>
Fep>Show Version
Fep version 22 running in prom.
NIL
```

si:halt with optional *fep-commands* can be included in function definitions to define booting and loading procedures, as shown in the following examples:

```
(defun boot-macsyma ()                          ;boots using >Macsyma.boot
  (si:halt "Boot >Macsyma.boot
"))

(defun reboot ()                                ;boots using the default
  (si:halt (format nil "Boot~%")))              ;boot file
```

```
(defun reload-world ()                                    ;loads the default world
  (si:halt (format nil "Load World~%Start~%")))           ;and starts Lisp

(defun reload-microcode ()                                ;loads the default
  (si:halt (format nil "Load Microcode~%Start~%")))       ;microcode and starts
```

si:machine-model *Function*

This function returns a keyword symbol designating the model number of the current 3600-family computer.

Possible return values are as follows:

:unknown	The model number cannot be determined (usually indicating lack of some ID prom)
:/3600 or :|3600|	(The keyword whose print-name is "3600".) The machine is a Symbolics 3600.
:/3670	The machine is a Symbolics 3670.
:/3640	The machine is a Symbolics 3640.

si:show-configuration *Function*

This function displays the hardware configuration of the current machine. For example:

```
(si:show-configuration)
NanoFEP (P.N. 170018) S.N. 382, manufactured on 1983-08-16
  Machine serial number 0.
  Manufactured as rev 1, functions as rev 1, ECO level 0
Datapath (P.N. 170032) S.N. 100, manufactured on 1983-04-05
  Manufactured as rev 3, functions as rev 3, ECO level 0
Sequencer (P.N. 170042) S.N. 100, manufactured on 1984-01-05
  Manufactured as rev 4, functions as rev 4, ECO level 0
Memory Control (P.N. 170052) S.N. 1254, manufactured on 1983-09-21
  Manufactured as rev 5, functions as rev 5, ECO level 0
Front End (P.N. 170062) S.N. 167, manufactured on 1983-03-18
  Manufactured as rev 5, functions as rev 5, ECO level 0
512K Memory (P.N. 170002) S.N. 511, manufactured on 1983-09-29
  Manufactured as rev 2, functions as rev 2, ECO level 0
  LBUS slot 0 (octal base address 0)
512K Memory (P.N. 170002) S.N. 588, manufactured on 1983-10-14
  Manufactured as rev 2, functions as rev 2, ECO level 0
  LBUS slot 6 (octal base address 14000000)
IO (P.N. 170082) S.N. 228, manufactured on 1983-05-31
  Manufactured as rev 2, functions as rev 2, ECO level 0
  LBUS slot 8 (octal base address 20000000)
FEP Paddle Card (P.N. 170066) S.N. 240, manufactured on 1983-06-02
  Manufactured as rev 1, functions as rev 1, ECO level 0
IO Paddle Card (P.N. 170086) S.N. 334, manufactured on 1983-09-12
  Ethernet address: 08-00-05-01-70-0A
  Monitor-type: Philips
  Manufactured as rev 1, functions as rev 1, ECO level 0
NIL
```

Appendix B
System Conventions and Helpful Hints

B.1 Miscellaneous Conventions

All uses of the phrase "Lisp reader", unless further qualified, refer to the part of Lisp that reads characters from I/O streams (the **read** function), and not the person reading this documentation.

By default, Symbolics-Lisp displays numbers in base 10. If you wish to change it: See the section "What the Reader Recognizes" in *Reference Guide to Symbolics-Lisp.*

Several terms that are used widely in other references on Lisp are not used much in Symbolics documentation, as they have become largely obsolete and misleading. They are: "S-expression", which means a Lisp object; "Dotted pair", which means a cons; and "Atom", which means, roughly, symbols and numbers and sometimes other things, but not conses. For definitions of the terms "list" and "tree": See the section "Manipulating List Structure" in *Reference Guide to Symbolics-Lisp.*

B.2 Answering Questions the System Asks

The system occasionally asks you to confirm some command. There are two forms this can take:

- Simple commands such as Load File or Save File Buffers might ask you to confirm with a question requiring a Y (for yes) or an N (for no).

  ```
  Save Buffer program.lisp >kjones>new-project> tuna: ? (Y or N)
  ```

  You press `Y` or `SPACE` for yes, `N` for no.

- Destructive commands, such as Initialize Mail, require that you type the entire word yes to confirm them.

  ```
  Do you really want to do this? (Yes or No)
  ```

  You must type the entire word yes to confirm the the command. Thus you are less likely to issue such a command accidentally.

Lisp provides several functions for this kind of querying: See the section "Querying the User" in *Programming the User Interface.*

B.3 Questions Users Commonly Ask

What is a Logical Pathname?

A logical pathname is a kind of pathname that doesn't correspond to any particular physical file server or host. Logical pathnames are used to make it easy to keep software on more than one file system. An important example is the software that constitutes the Lisp Machine system. Every site has a copy of all of the sources of the programs that are loaded into the initial Lisp environment. Some sites might store the sources on a UNIX file system, while others might store them on a TOPS-20. However, the software needs to find these files no matter where they are stored. This is accomplished by using a logical host called SYS. All pathnames for system software files are actually logical pathnames with host SYS. At each site, SYS is defined as a logical host, and there is a translation table that maps the SYS host to the actual physical machine for that site.

Here is how translation is done. For each logical host, there is a mapping that takes the name of a directory on the logical host, and produces a device and a directory for the corresponding physical host. For example, the logical host SYS has a directory SITE;. At a site that keeps its sources on a TOPS-20 this might map to SS:<SITE> . Then the file SYS:SITE;NAMESPACE.LISP translates to SS:<SITE>NAMESPACE.LISP. On a UNIX system this same file might translate to /usr/system/namespace.l. The important thing is that everyone can refer to the file by its logical pathname, SYS:SITE;NAMESPACE.LISP, where the name before the ":" is the logical host name, and logical directories are separated by ";"s. You can define the translation of a logical pathname to be any physical pathname of any operating system type, but to access a file with a logical pathname you need only to use logical pathname syntax.

The function **fs:set-logical-pathname-host** is used to define a logical host and its logical directories. Here are some sample uses:

```
(fs:set-logical-pathname-host "SYS" :physical-host "my-vms"
                    :translations '(("games;" "[games]")
                                    ("*;" "[symbolics.*]")))
```

This says that sys:games; translates to my-vms:[games], and that any other logical directory on the logical host SYS translates to a subdirectory under [symbolics] of the same name. See the function **fs:set-logical-pathname-host** in *Reference Guide to Streams, Files, and I/O.*

What is a World Load?

A world load can be thought of as a snapshot of an operating Lisp environment. All of the functions, variables, and other Lisp objects that were present in the Lisp environment when the snapshot was made are contained in the world load file on the disk. Typically, snapshots of worlds are made only when such a snapshot would save significant time later. For example, after you have initially configured your new machine at your site, it is useful to make a snapshot of the configured environment because it saves you time in the future (you don't have to configure the machine each time you boot it). If you usually load MACSYMA or FORTRAN each time you boot, it is advantageous to make a snapshot of a world with that software loaded, to save you the time of loading it. Remember, everything in the environment is contained in the snapshot, so you don't want to create a world load file after you've been using the editor or most system facilities (you don't want to find old text in your editor buffer when you cold boot.). The way to create a snapshot and save it to disk is by using the command Save World or the function **(disk-save)**.

Why Do You Name Machines and Printers?

Naming inanimate objects such as hosts, printers, sites, and networks may seem foolish if you have one or fewer of each, but if you have large numbers of machines, names are a convenient way to easily refer to a particular machine with a particular address without having to remember its network address, machine type, and physical location. One customer named its machines after the characters in Winnie the Pooh, while another named its after the wives of Henry VIII.

B.4 Questions About the FEP and LMFS

Why Can't I Write Out Files When I Have Free Disk Space?

The 3600 disk is physically divided into partitions known as FEP files. This division of the disk is called the FEP file system. However, when one speaks of the file system of a Lisp Machine, one is generally referring to the LMFS (Lisp Machine File System) of that machine. This is the file system you edit when you click left on [Tree Edit Root] in the FSEdit window, and is the file system used when you specify file names of the form *Lisp Machine Name:>directory>filename.type.version*. The entire Lisp Machine local file system normally resides inside one big file of the FEP file system (typically FEP0:>LMFS.FILE.1). Thus, LMFS is full when the amount of space allocated to it (in other words, FEP0:>LMFS.FILE.1) is full. Thus, LMFS could be full but there could still be 100,000 unused blocks on the disk (not even allocated as FEP files). See the section "Adding a Spare World Load as LMFS File Space", page 212.

How Do I Create a FEP File?

There aren't too many reasons for creating FEP files. If you want to create a file to allocate more LMFS file space, simply enter the File System Editor window (FSEdit), by using SELECT F, by clicking on [File System] in the System Menu, or by using the Select Activity FSMaint command. Then click on [Local FS Maint]. Click right on [Initialize]. A menu pops up. Click on [Auxiliary Partition] and click on the name above this so that you can specify a name for the auxiliary partition. Typically, a good name is FEP0:>LMFS-AUX.FILE. (Of course, if you have more than one drive, or a FEP file named LMFS-AUX.FILE already exists, you should choose another name.) Then click on [Do It]. It will ask you how much space to allocate to this file; specify a number of blocks.

When working with FEP files, the File System Editor is good only for creating FEP files to be allocated to LMFS. If you need a FEP file for another purpose (extra paging, for example) and create it with FSEdit, the LMFS data structure contained on your disk might become very confused, and can potentially destroy the file system of your machine. The following Lisp form creates a FEP file for purposes other than a LMFS partition.

```
(WITH-OPEN-FILE (FILE FEPn:>Filename.type.version
                      :DIRECTION :BLOCK
                      :IF-EXISTS :ERROR)
   (SEND FILE :GROW 30000))
```

The italicized string above represents the name of the FEP file to be created, and the italicized 30000 represents the size you want to make the file.

For more information about LMFS and the FEP file system: See the section "FEP File System Overview", page 208.

Appendix C
Documentation Notation Conventions

C.1 Understanding Notation Conventions

You should understand several notation conventions before reading the documentation.

C.1.1 Lisp Objects

C.1.1.1 Functions

A typical description of a Lisp function looks like this:

function-name *arg1 arg2* &optional *arg3 (arg4* **(foo 3)***)* *function*

Adds together *arg1* and *arg2*, and then multiplies the result by *arg3*. If *arg3* is not provided, the multiplication is not done. **function-name** then returns a list whose first element is this result and whose second element is *arg4*. Examples:

```
(function-name 3 4) => (7 4)
(function-name 1 2 2 'bar) => (6 bar)
```

The word "&optional" in the list of arguments tells you that all of the arguments past this point are optional. The default value can be specified explicitly, as with *arg4*, whose default value is the result of evaluating the form **(foo 3)**. If no default value is specified, it is the symbol **nil**. This syntax is used in lambda-lists in the language. (For more information on lambda-lists: See the section "Evaluating a Function Form" in *Reference Guide to Symbolics-Lisp*.) Argument lists can also contain "&rest", which is part of the same syntax.

Note that the documentation uses several *fonts*, or typefaces. In a function description, for example, the name of the function is in boldface in the first line, and the arguments are in italics. Within the text, printed representations of Lisp objects are in the same boldface font, such as **(+ foo 56)**, and argument references are italicized, such as *arg1* and *arg2*.

Other fonts are used as follows:

Fixed-width font `(function-name)`
For user input or Lisp examples that are set off from the text

"Key" font (RETURN, c-L)
For keystrokes in running text

C.1.1.2 Macros and Special Forms

The descriptions of special forms and macros look like this:

do-three-times *form* *Special Form*

Evaluates *form* three times and returns the result of the third evaluation.

with-foo-bound-to-nil *form...* *Macro*

Evaluates the *forms* with the symbol **foo** bound to **nil**.
It expands as follows:

```
(with-foo-bound-to-nil
form1
form2 ...) ==>
(let ((foo nil))
form1
form2 ...)
```

Since special forms and macros are the mechanism by which the syntax of Lisp is extended, their descriptions must describe both their syntax and their semantics; unlike functions, which follow a simple consistent set of rules, each special form is idiosyncratic. The syntax is displayed on the first line of the description using the following conventions.

- Italicized words are names of parts of the form that are referred to in the descriptive text. They are not arguments, even though they resemble the italicized words in the first line of a function description.

- Parentheses ("( )") stand for themselves.

- Square brackets ("[ ]") indicate that what they enclose is optional.

- Ellipses ("...") indicate that the subform (italicized word or parenthesized list) that precedes them can be repeated any number of times (possibly no times at all).

- Curly brackets followed by ellipses ("{ }...") indicate that what they enclose can be repeated any number of times. Thus, the first line of the description of a special form is a "template" for what an instance of that special form would look like, with the surrounding parentheses removed.

The syntax of some special forms is too complicated to fit comfortably into this style; the first line of the description of such a special form contains only the name, and the syntax is given by example in the body of the description.

The semantics of a special form includes not only its contract, but also which subforms are evaluated and what the returned value is. Usually this is clarified with one or more examples.

A convention used by many special forms is that all of their subforms after the first few are described as "*body*...". This means that the remaining subforms constitute the "body" of this special form; they are Lisp forms that are evaluated one after another in some environment established by the special form.

This imaginary special form exhibits all of the syntactic features:

twiddle-frob *[(frob option...)] {parameter value}...* *Special Form*

Twiddles the parameters of *frob*, which defaults to **default-frob** if not specified. Each *parameter* is the name of one of the adjustable parameters of a frob; each *value* is what value to set that parameter to. Any number of *parameter/value* pairs can be specified. If any *options* are specified, they are keywords that select which safety checks to override while twiddling the parameters. If neither *frob* nor any *options* are specified, the list of them can be omitted and the form can begin directly with the first *parameter* name.

frob and the *values* are evaluated; the *parameters* and *options* are syntactic keywords and are not evaluated. The returned value is the frob whose parameters were adjusted. An error is signalled if any safety check is violated.

C.1.1.3 Methods and Variables

Methods, the message-passing equivalent of ordinary Lisp's functions, are described in this style:

message-name *arg1 arg2* &optional *arg3* (of **flavor-name**) *Method*

This is the documentation of the effect of sending a message named **message-name**, with arguments *arg1*, *arg2*, and *arg3*, to an instance of flavor **flavor-name**.

Descriptions of variables ("special" or "global" variables) look like this:

typical-variable *Variable*

The variable **typical-variable** has a typical value....

C.1.2 Macro Characters

The characters acute accent (') (also called the single quote character) and semicolon (;) have special meanings when typed to Lisp; they are examples of what are called *macro characters*. It is important to understand their effect.

When the Lisp reader encounters a single quote, it reads in the next Lisp object and encloses it in a **quote** special form. That is, **'foo-symbol** turns into **(quote foo-symbol)**, and **'(cons 'a 'b)** turns into **(quote (cons (quote a) (quote b)))**. The reason for this is that "**quote**" would otherwise have to be typed in very frequently and would look ugly.

Note that, in Lisp, *quoting* a character means inhibiting what would otherwise be special processing of it. Thus, the character "/" is used for quoting unusual characters so that they are not interpreted in their usual way by the Lisp reader, but rather are treated the way normal alphabetic characters are treated. So, for example, in order to give a "/" to the reader, you must type "//", the first "/" quoting the second one. When a character is preceded by a "/" it is said to be *slashified*. Slashifying also turns off the effects of macro characters such as single quote and semicolon.

The following characters also have special meanings, and cannot be used in symbols without slashification. These characters are explained in detail elsewhere: See the section "Printed Representation" in *Reference Guide to Symbolics-Lisp.*

"	Double-quote delimits character strings.
#	Number-sign introduces miscellaneous reader macros.
‘	Backquote is used to construct list structure.
,	Comma is used in conjunction with backquote.
:	Colon is the package prefix.
|	Characters between pairs of vertical bars are quoted.
⊗	Circle-X lets you type in characters using their octal codes.

The semicolon is used as a commenting character. When the Lisp reader sees one, the remainder of the line is discarded.

C.1.3 Character Case

All Lisp code in the documentation is written in lowercase. In fact, the reader turns all symbols into uppercase, and consequently everything prints out in uppercase. You can write programs in whichever case you prefer.

C.1.4 Packages and Keyword Names

Various symbols have the colon (:) character in their names. By convention, all *keyword symbols* in the system have names starting with a colon. The colon character is not actually part of the print name, but is a package prefix indicating that the symbol belongs to the package with a null name, which means the **keyword** package. (For more information on colons: See the section "Introduction to Keywords" in *Reference Guide to Symbolics-Lisp.* For now, just pretend that the colons are part of the names of the symbols.)

The document set describes a number of internal functions and variables, which can be identified by the "**si:**" prefix in their names. The "**si**" stands for "**system-internals**". These functions and variables are documented because they are things you sometimes need to know about. However, they are considered internal to the system and their behavior is not as guaranteed as that of everything else.

C.1.5 Maclisp

Symbolics-Lisp is descended from Maclisp; throughout the documentation, there are notes about differences between the dialects. For the new user, it is important to note that some Symbolics-Lisp functions exist solely for Maclisp compatibility; they should *not* be used in new programs. Such functions are clearly marked in the text.

C.1.6 The Character Set

The Symbolics 3600-family character set is not the same as the ASCII character set used by most operating systems. For more information: See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O.*

Unlike ASCII, there are no "control characters" in the character set; Control and Meta are merely things that can be typed on the keyboard.

C.2 Notation Conventions Quick Reference

Modifier Key Conventions

Modifier keys are designed to be held down while pressing other keys. They do not themselves transmit characters. A combined keystroke like META-X is pronounced "meta x" and written as m-X. This notation means that you press the META key and, while holding it down, press the X key.

Modifier keys are abbreviated as follows:

Key	*Abbreviation*
CTRL	c-
META	m-
SUPER	s-
HYPER	h-
SHIFT	sh-
SYMBOL	sy-

The keys with white lettering (like X or SELECT) all transmit characters. Combinations of these keys are meant to be pressed in sequence, one after the other. This sequence is written as, for example, SELECT L. This notation means that you press the SELECT key, release it, and then press the L key.

Documentation Conventions

This documentation uses the following notation conventions:

Appearance in document *Representing*

send, chaos:host-up	Printed representation of Lisp objects in running text.
`RETURN`, `ABORT`, `c-F`	Keyboard keys.
`SPACE`	Space bar.
`login`	Literal type in.
`(make-symbol "foo")`	Lisp code examples.
(**function-name** *arg1* <u>*arg2*</u>)	Syntax description of the invocation of **function-name**.
arg1	Argument to the function **function-name**, usually expressed as a word that reflects the type of argument (for example, *string*).
<u>*arg2*</u>	Optional argument; you can leave it out.
Undo, Reply, Start	Command Processor command names and command names in Zmacs, Zmail, and the front-end processor (FEP) appear with the initial letter of each word capitalized.
Insert File (`m-X`)	Extended command names in Zmacs and Zmail. Use `m-X` to invoke one.
[Map Over]	Menu items.
(L), (R2)	Mouse clicks: L=left, L2=`sh`-left, M=middle, M2=`sh`-middle, R=right, R2=`sh`-right. (`sh`-left means that you press the `SHIFT` key while holding down the left mouse button. You can achieve the same result by clicking the button quickly twice.)

Mouse Command Conventions

The following conventions are used to represent mouse actions:

1. Square brackets delimit a menu item.

2. Slashes (/) separate the members of a compound mouse command.

3. The standard clicking pattern is as follows:

   - For a single menu item, always click left. For example, the following two commands are identical:

     [Previous]
     [Previous (L)]

   - For a compound command, always click right on each menu item (to display a submenu) except the last, where you click left (to cause an action to be performed). For example, the following two compound commands are identical:

     [Map Over / Move / Hardcopy]
     [Map Over (R) / Move (R) / Hardcopy (L)]

4. When a command does not follow the standard clicking order, the notation for the command shows explicitly which button to click. For example:

   [Map Over / Move (M)]
   [Previous (R)]

Index

	HELP V Zmacs command 51
	HELP W Zmacs command 51
Show	Herald Command 37
System Conventions and Helpful	Hints 227
	Hints on Using the FEP 197
Command	History 15
Setting Mode	Hooks in Init Files 170
Delete	Horizontal Space 53
Display status of	hostat 185
Quitting	Hostat 183
	hostat function 183
	Hostat (H) 188
	Host object 112
	Hosts 113
Show	Hosts Command 37
	Host Status 37, 183
	Host status report 183
	How Do I Create a FEP File? 229
	How LMFS Uses the FEP File System 209
	How the Input Editor Works 101
	How the Inspector Works 189
	How to Change Fonts 75
	How to Create an Init File 164
	How to Get Hardcopy 107
When and	How to Use the Garbage Collector 105

SELECT	I 189
Set User	Id Command 34
Functions for	identifying callers 177
	Identifying callers of variables 177
	Image restore format 201
Logging	in 1, 155
Lookup Commands Available	in an Editor 128
	Inbox 64
Zmail: Get	Inbox 64
	Incoming messages 70
Delete	Indentation 53
Loading documentation	index files 121
	Index of Function Keys 139
Introduction:	Index of Function Keys 139
Attribute	indicator 112
^F font change	indicators 69
Summary of Help Functions	in Different Contexts 86
Comments	in FEP commands 197
Saving terminal interactions	in file 183
Disk label	information 200, 213
Zmail Mode Line	Information 61
Setting Editor Variables	in Init Files 167
Setting Key Bindings	in Init Files 168
Setting Mode Hooks	in Init Files 170
Zmacs Customization	in Init Files 167
How to Create an	Init File 164
Logging in Without Processing Your	Init File 164
	Init File Form: Auto Fill in Text Mode 170
	Init File Form: Balanced Quotation Marks and Asterisks 168

Document Examiner List	of Current Candidates 131
List	of current candidates 127
Description	of Erasing Text 53
Summary	of Erasing Text 53
Turning the Command Processor on and	Off 15
Turning	off Converse messages 72
Display status	of file system display 185
Use	of Fonts in the Documentation 231
Formatted print	of function definitions 180
Index	of Function Keys 139
Introduction: Index	of Function Keys 139
Overview	of Garbage Collection 105
Checking the Status	of Hardcopy Devices 109
Reference Description	of Help Functions 88
Summary	of Help Functions in Different Contexts 86
Display status	of hostat 185
Beginning	of Line 50
End	of Line 50
Quick Summary	of Mouse Functions 149
Description	of Moving the Cursor 50
Summary	of Moving the Cursor 50
Data Types	of Namespace System Attributes 112
Overview	of Peek 185
Status	of registers 200
Example	of Setting Fonts in the Input Editor 79
Overview	of the Command Processor 9
Beginning	of Topic 134
End	of Topic 134
Identifying callers	of variables 177
Display status	of window area 185
Overview	of Zmacs 47
Zmacs Commands for Finding Out About the State	of Zmacs 86
Turning the Command Processor	on and Off 15
Turning	on Converse messages 72
Init File Form: Edit Buffers	on c-X c-B 169
Using the	Online Documentation System 119
Init File Form: m-.	on m-(L) 169
Init File Form: Edit Buffers	on m-X 169
Init File Form: c-m-L	on the SQUARE Key 169
Hints	on Using the FEP 197
Checking	on What the Machine Is Doing 177
	&optional 231
Debugger Proceed and Restart	Options 193
Profile mode	options 170
Setting and Saving Zmail	Options 171
User	options window 171
HELP	or c-? 52
Init File Form:	Ordering Buffer Lists 168
Subject	or Text Zmail display column heading 57
Completion in	Other Contexts 85
Talking to	Other Users 69
Cold boot after logging	out 3
Logging	Out 3
Zmacs Commands for Finding	Out About Flavors 87
Lisp Facilities for Finding	Out About Lisp 88
Zmacs Commands for Finding	Out About Lisp 86
Zmacs Commands for Finding	Out About the State of Buffers 86
Zmacs Commands for Finding	Out About the State of Zmacs 86