Genera User's Guide

Overview of Symbolics Computers

Documentation Notation Conventions

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:

CONTROL	c-
META	m-
SUPER	s-
HYPER	h-
SHIFT	sh-
SYMBOL	sy-

Modifier keys can be used in combination, as well as singly. For example, the notation c-m-Y indicates that you should hold down both the CONTROL and the META keys while pressing Y.

Modifier keys can also be used, both singly and in combination, to modify mouse commands. For example, the notation sh-Left means hold down the SHIFT key while clicking Left on the mouse and c-m-Middle means hold down CONTROL and META while clicking Middle.

The keys with white lettering (like X or SELECT) all transmit characters. Combinations of these keys should be pressed in sequence, one after the other (for example, SELECT L). This notation means that you press the SELECT key, release it, and then press the L key.

LOCAL is an exception to this rule. Despite its white lettering, you must hold it down while pressing another key, or it has no effect. For example, to brighten the image on your monitor, you would hold down LOCAL while pressing B.

Documentation Conventions

This documentation uses the following notation conventions:

cond, **zl:hostat**	Printed representation of Lisp objects in running text.
RETURN, ABORT, c-F	keys on the Symbolics Keyboard.
SPACE	Space bar.
login	Literal typein.
(make-symbol "foo")	Lisp code examples.
(**function-name** *arg1* &optional *arg2*)	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*).
&optional	Introduces optional argument(s).
Show File, Start	Command Processor command names and Front-end Processor (FEP) command names appear with the initial letter of each word capitalized.
m-X Insert File, Insert File (m-X)	Extended command names in Zmacs, Zmail, and Symbolics Concordia appear with the m-X notation either preceding the command name, or following it in parentheses. Both versions mean press m-X and then type the command name.
[Map Over]	Menu items. Click Left to select a menu item, unless other operations are indicated. (See the section "Mouse Command Conventions".)
Left, Middle, Right	Mouse clicks.
sh-Right, c-m-Middle	Modified mouse clicks. For example, sh-Right means hold down the SHIFT key while clicking Right on the mouse, and c-m-Middle means hold down CONTROL and META while clicking Middle.

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 equivalent:

     [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)]

Introduction to Genera

The software environment that runs on the Symbolics family of computers is called Genera. See the document *Genera Concepts*.

The Console

The devices that are used to talk to Genera are collectively referred to as the *console*. These include one or more bit-raster displays, a specially extended keyboard, and a pointing device called a *mouse*.

The Screen

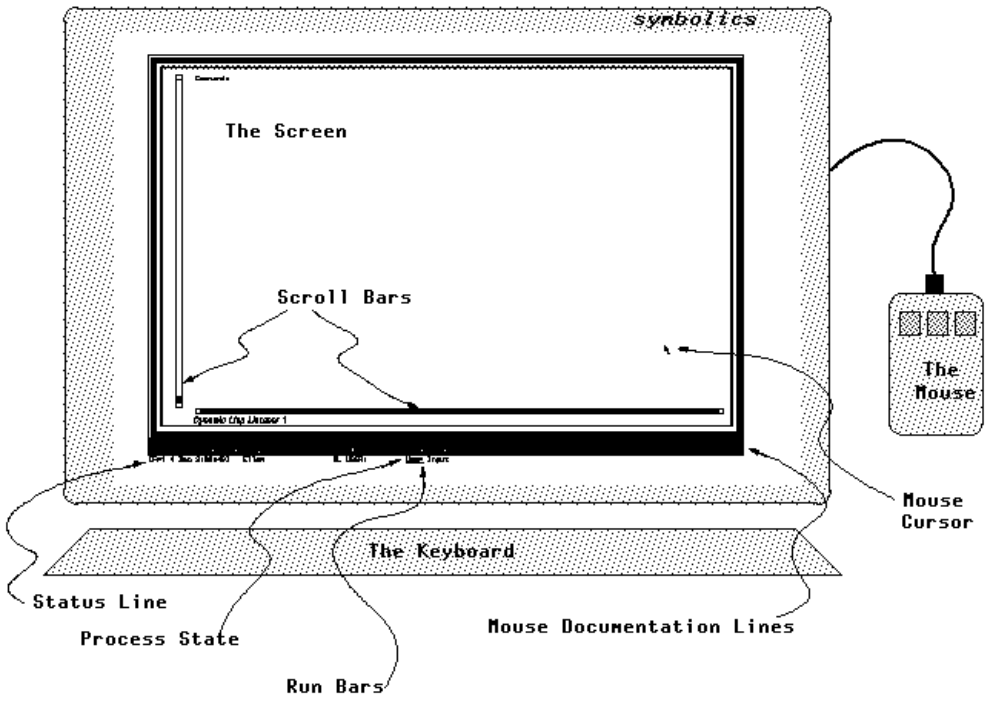


Figure 132. The Console

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 tell you what your machine is doing, what mouse actions are available, and are helpful in determining whether Genera is operating normally or needs intervention. See the section "Recovering From Errors and Stuck States".

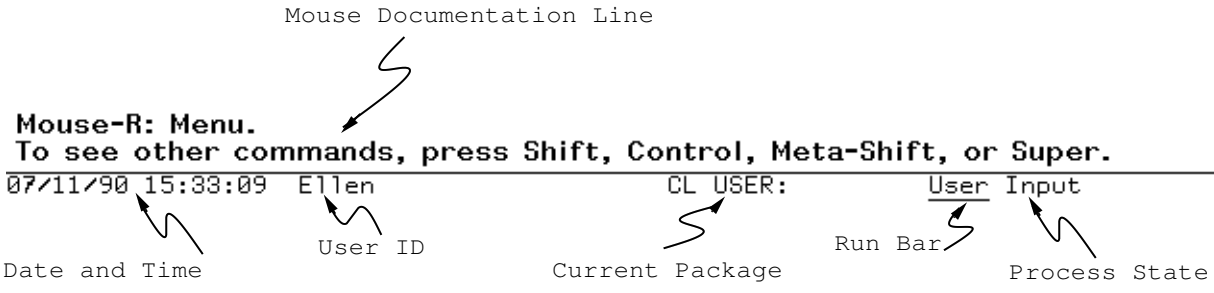


Figure 133. The Status Area

Mouse Documentation Lines

The mouse documentation lines contain information about what different mouse clicks mean. As you move the mouse across different mouse-sensitive areas of the screen, the mouse documentation lines change 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 lines. When the mouse documentation lines are blank at "top level" in a window, the mouse usually offers some standard commands. Clicking Left selects a window. Clicking Right often brings up a menu specific to the application. Clicking sh-Right brings up the System menu.

The mouse documentation lines are normally displayed in white type in a black box (reverse video). Pressing FUNCTION m-C reverses the video state of the mouse documentation lines; if they were white on black, they become black on white and vice versa.

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 current information, such as
  - Console idle time
  - Network service indicators

Process State

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

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.
User Input	Waiting for input from keyboard or mouse.

Run Bars

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

Two GC bars (under the package name)
: The left one is visible when the scavenger is looking for references to objects that are candidates to become garbage. The right one is visible when the transporter is copying an object. See the section "Theory of Operation of the GC Facilities".

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. See the section "How the Scheduler Works".

Paging Bar
: Visible when paging activity is going on.

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

Progress Notes

Progress notes for some activities are displayed at the right end of the status line. They consist of a string describing what is happening — the pathname of file being written or read, for example — and a bar that progresses toward the right margin according to how far the process has gotten. Progress notes can be added to your application programs: See the section "Progress Indicator Facilities".

The Keyboard

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

The keys are divided into three groups: special function keys, character keys, and modifier keys. Special function keys and character keys transmit something. They have black labels 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 red labels.

Function Keys
: FUNCTION, ESCAPE, REFRESH, CLEAR INPUT, SUSPEND, RESUME, ABORT, NETWORK, HELP, 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	LOCAL, CAPS LOCK, SYMBOL, SHIFT, REPEAT, MODE LOCK, HYPER, SUPER, META, and CONTROL

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

```
CIRCLE
SQUARE
TRIANGLE
HYPER
```

To get information about the keyboard, press the SYMBOL and HELP keys simultaneously.

You can have keys repeat if they are held down. This feature is disabled by default, but you can enable it by setting **si:*kbd-auto-repeat-enabled-p*** to **t**.

```
(setf si:*kbd-auto-repeat-enabled-p* t)
```

The speed of repetition is controlled by **si:*kbd-repetition-interval***. See the variable **si:*kbd-repetition-interval***.

You can exempt certain keys from auto-repetition using the function **si:set-auto-repeat-p**. For example, to make SQUARE one of the keys that do not auto-repeat, you would type:

```
(si:set-auto-repeat-p #\Square nil)
```

See the function **si:set-auto-repeat-p**.

You can customize key bindings; see the section "Setting Key Bindings in Init Files".

If you have a MacIvory or a UX400S, there is a mapping from those keyboards to the Symbolics special keys. For the MacIvory, see the section "Using the Genera Application on a MacIvory". For the UX400S, see the section "Symbolics UX Keyboard Templates".

The Mouse

The mouse is a pointing device that can be moved around on a flat surface. These motions are sensed by Genera, 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. Typically you point at something with the mouse and specify an operation by clicking the mouse buttons. "Chorded clicks", indicated by a modifier key (c-, m-, c-m, sh-, c-sh, m-sh, and s-) perform different actions than single clicks. In any specific context, there are up to 96 operations that can be performed with the mouse, invoked by Left, Middle, Right, and chorded clicks. Some of these operations are lo-

cal 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 in the mouse documentation lines at the bottom of the screen. The mouse documentation lines change as you move the mouse around or run different programs.

Sometimes holding a mouse button down continuously for a period of time is also defined to perform some operation, for example, drawing a curve on the screen. This is indicated by the word "Hold". For example, "Middle Hold" means to press 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 press the left button, then release the SHIFT key but hold the left button down until the operation is complete.

The Mouse and Menus

Mouse-sensitivity

Parts of the screen can be *mouse-sensitive;* that is, clicking one of the mouse buttons on these areas causes some action to occur. When the mouse cursor moves over a portion of the screen that is mouse sensitive, an outline box appears around the item. Clicking on the boxed item in the manner specified in the mouse documentation line causes the desired action to occur.

Scrolling

Many windows in the system respond to scrolling commands. On the left side of the pane is a scroll bar. When the mouse is moved over a scroll bar, its cursor changes to a double-headed pointer.

The gray area in the bar indicates the percentage of the pane or buffer contents that is currently visible on the screen.

To scroll using the scroll bar and the double-headed pointer, use one of the following mouse buttons:

Left	Moves the line indicated by the pointer to the top of the screen.
sh-Left	Moves the line indicated by the pointer to the bottom of the screen.
Middle	Displays the percentage of the pane contents that approximately corresponds to the position indicated by the pointer.
Right	Moves the line currently at the top of the screen to the position indicated by the pointer.

Some window can be scrolled horizontally. See the section "Scrolling with the Mouse". If you do not like having the scroll bar on the left, you can change its location. See the section "Adjusting Console Parameters".

Menus

One common application of a mouse button is to call up a *menu* of options. Menus are lists of mouse sensitive choices, surrounded by a border. They normally appear in the part of the screen where the mouse cursor was positioned when you clicked the button.

Genera has several styles of menus, including the following common ones:

- Momentary menu

  Each item is a possible choice. Positioning the mouse cursor over an item and then clicking the appropriate button makes the choice. Momentary menus disappear after you make a choice or when you move the mouse off them. The System menu (shown in Figure 61) is a momentary menu.

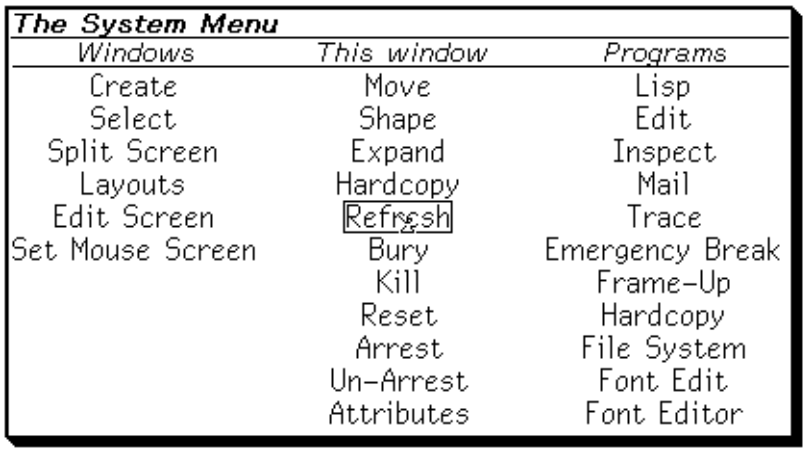


Figure 134. A Momentary Menu

- Accept-variable-values menu

  Each line presents one or more possible values of a particular variable. You select the variables and values you want. An accept-variable-values remains visible until you click on the end or exit button. The Hardcopy menu (shown in Figure 66) is a accept-variable-values menu.
  Each variable has a type that controls what values it can take on. The way in which the possible values are presented and the way in which you choose a value depend upon the type. Variables can have one of two types.

  ° A type with a small number of valid values. Each line in the menu presents the possible valid values of a particular variable. The current value appears in bold face. *Orientation* is such a variable. Its valid values are Landscape and Portrait. Each of the values is mouse-sensitive. Clicking on a value selects it.

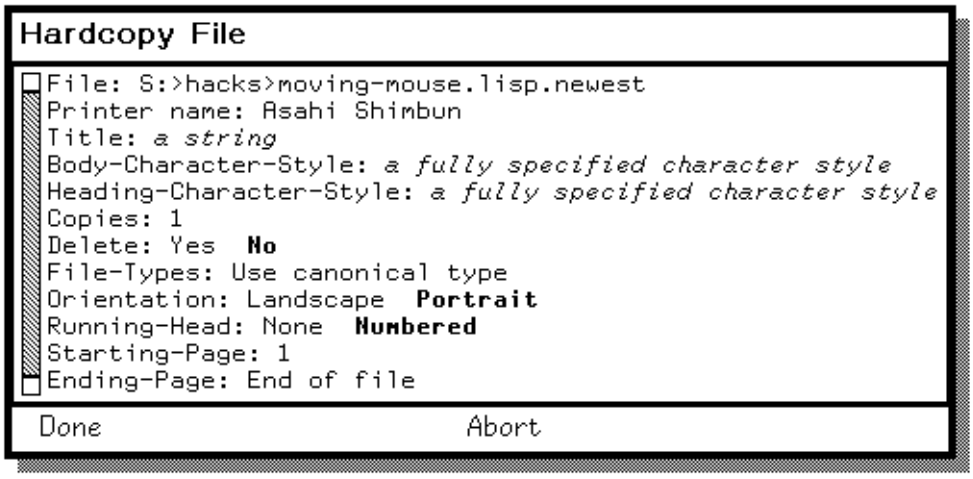


Figure 135. An Accept-variable-values Menu

° A type with a large or infinite number of legal values. Each line in the menu presents only the current value of a particular variable. *File*, *Printer*, *Title*, and *Body-character-style* are variables of this type. To change a value, select the current value by clicking on it, type in a new value, and press RETURN. Rubbing out more characters than have been typed in restores the original value instead of changing it.

You exit menus in a variety of ways. For momentary menus, such as the System menu, making the choice causes the menu to disappear. Moving the mouse cursor off this kind of menu also causes the menu to disappear. Other menus have explicit commands, such as [Do It], [Done], [End], [Exit], or [Abort], which you must click on to make the menu disappear. Other menus are displayed in the frame permanently, such as the Zmail Command Menu.

System Menu

The System menu is a momentary menu that lists several choices for acting upon windows and calling programs (for example, Lisp Listener, Zmacs, or the Inspector). (See Figure 134.) You can always call the System menu by clicking sh-Right. Use the System menu to do many things, among them:

- Create new windows.
- Select old windows.
- Change the size and placement of windows on the screen.
- Hardcopy a file.

For more information about using the System menu, see the section "Using the System Menu".

For more information about the mouse and menus, see the section "Using the Window System" and see the section "Window System Choice Facilities".

Selecting and Creating Activities

The programs that run under Genera are called *activities*. An activity can have one or more *windows*. All user interaction with Genera, occurs in windows. The window that you are interacting with is called the *selected window*. You select a window via the mouse, a menu, or a keyboard key. If the window is not already exposed, it appears on the screen. See *Programming the User Interface* for more information about windows and how you can create them for your applications.

Standard Activities

Genera has a standard set of activities available, some of which are available via the System menu and all of which are available via the SELECT key. Press SELECT HELP for a complete list of activities available in your world. The activities available via the SELECT key and the characters to use to select them are:

=	Select Key Selector
C	Converse
D	Document Examiner
E	Editor
F	File System Maintenance Operations
I	Inspector
L	Lisp
M	Zmail
N	Notifications
P	Peek
Q	Frame-Up
T	Terminal
X	Flavor Examiner

Some activities are initialized in the world when it is built. Others are initialized when they are selected for the first time. Click on [Select] in the Windows column of the System menu (see Figure 134) for a menu of the current initialized windows. If you have more than one copy of an activity, both are listed:

```
Dynamic Lisp Listener 1
Main Zmail Window 1
Zmacs Frame 1
Dynamic Lisp Listener 2
```

Moving

In Genera you do not "leave" an activity with an explicit terminating command; instead, you select a different activity. You can return to the most recently used activity by pressing FUNCTION S.

Starting Up

This section provides information about how to start, cold boot, log in to, and log out of the 3600-family of machines and the XL400. 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 document *Genera 8.1 Software Installation Guide*.

Powering Up

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

1. Plug in the machine.

2. For the 3640, 3645, 3670, and 3675: Press the POWER button on the front panel.

   For the 3650: Turn the key on the front panel to PWR.

   For the 3620 and 3630: Press the ON button on the front panel.

   For the XL400: Press the power switch on the front panel; this is the rightmost switch. (Also, be sure the main power switch is turned on; this is located at the back of the machine, near the floor.)

If you have a 3600:

1. Plug in the 3600. The front panel lights on the processor cabinet display "3600" when the machine is plugged in. If they are not lit, check that the main circuit breaker at the lower rear of the cabinet is turned on.

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

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

Cold Booting After Powering Up

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

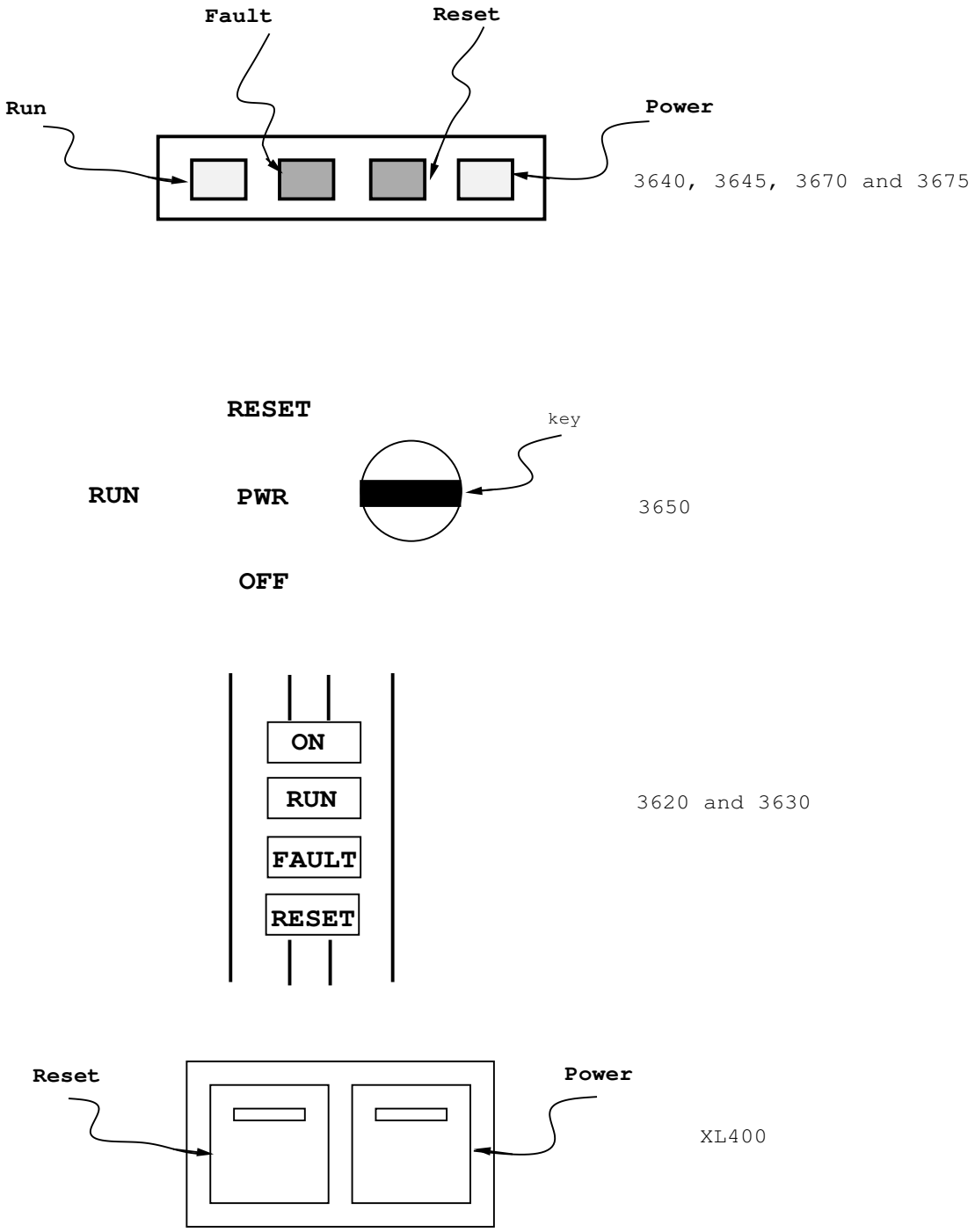


Figure 136. Front Panels of Symbolics Computers

Cold booting is a complete reset of Lisp. It loads in a fresh copy of Genera, called a *world load*. Cold booting erases any existing contents of Genera, 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 appears in the upper left-hand corner.
- A blinking cursor appears.

Figure ! shows what the screen looks like when the machine is ready to boot.

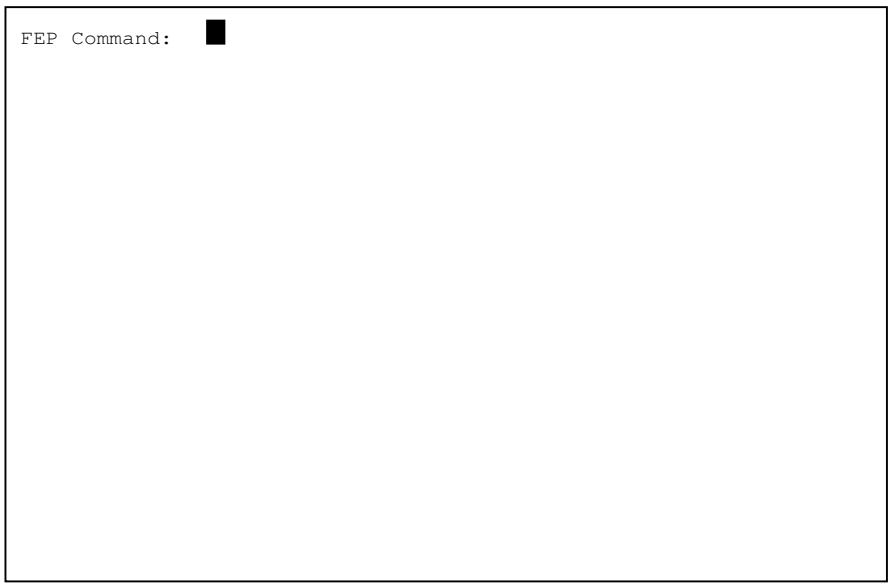


Figure 137. A Screen Ready to Boot

However, if either of the following conditions is true, you might not be able to cold boot:

- The screen is black. This might indicate that the console is not turned on. Check the switch on the rear of the console. If the console is turned on, press any key. The screen dims automatically when the console has been idle for 20 minutes. Pressing any key or moving the mouse brightens it. If the screen does not brighten when a key is pressed, press LOCAL B (hold down the LOCAL key while you press B). If holding down LOCAL B for several seconds fails to brighten the screen, your console may be malfunctioning. Call your Symbolics Field Service Representative.
- 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. Call your Symbolics Field Service Representative.

When you have determined that you can cold boot the machine, use the following procedure:

1. Type `Hello` and press `RETURN`.

   Several lines reporting files being scanned are displayed, followed by the FEP prompt. These overlay files initialize the microcode, identify paging files that are loaded during the boot process, and prepare the machine for booting Lisp.

2. Type `Boot` and 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 3650 System*** or ***Symbolics 3645 System*** for example) appears.

For more information about cold booting: See the section "The Front-End Processor".

Logging In

After cold booting, you are in a window named Dynamic Lisp Listener 1. You are now ready to log in. Logging in tells Genera 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.

If your login name is KJones, you can log in in any of the following ways. (Note that the examples are given in uppercase and lowercase, but the machine is not case sensitive. You can use all uppercase, all lowercase, or mixed case as you prefer.)

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

```
Login KJones
```

- To log into your machine, without your init file, type

```
Login KJones :init file none
```

- To log into another machine "sc3", using your init file, type

```
Login KJones :host sc3
```

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

For information about how to write init files, see the section "Customizing Genera".

Logging Out

1. Press SELECT L to get to a Lisp Listener.

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

   The Logout command cleans up and checks to see if you have left any unsaved buffers or mail files. If it finds any, it offers to save them for you.

   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.

Powering Down

To power down your machine:

1. Log out by giving the command:

   ```
   Logout
   ```

2. Halt the machine by giving the command:

   ```
   Halt Machine
   ```

3. For the 3640, 3645, 3670, and 3675: Give the shutdown command to the FEP:

   ```
   Shutdown
   ```

   or press the power button on the front panel.

   For the 3650: Turn the key on the front panel to OFF.

   For the 3620 and 3630: Press the ON button on the front panel.

   If you have a 3600, turn the key on the front panel to the OFF position.

Note: it is not necessary to turn off the circuit breaker on the back of the machine unless you are planning to unplug the machine and move it.

Getting Acquainted with Genera

This section offers a tutorial tour to introduce you to some of the features of the Genera environment. You probably want to sit down at a console and try things as you go along.

Using the System Menu

1. Press down the SHIFT key and click the right mouse button. (This is usually denoted in documentation as sh-Right.) You should see the System menu pop up on the screen as in Figure !.

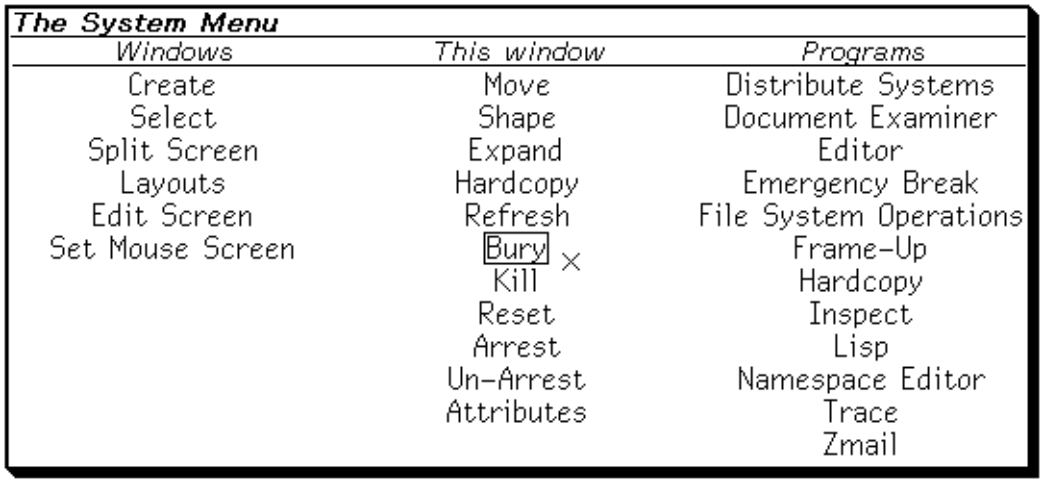


Figure 138. The System Menu

Notice that the mouse cursor has become an x and that there is a box around the word refresh. The word refresh is in the column titled "This Window" and its being in a box means that refresh is *mouse sensitive*. If you click a mouse button while the box is around refresh, thereby *selecting* it, the current window ("This window") would be *refreshed*, that is cleared and redrawn. The mouse documentation line at the bottom of the screen says:

```
Refresh the window the mouse is over.
```

Move your mouse so that it is off this menu. The menu disappears. That is the way *momentary* menus work: they disappear when the mouse is moved off them.

Click sh-Right again. Notice that the System menu again pops up, but it has followed your mouse. A momentary menu always pops up where your mouse is, because it can only remain on the screen when the mouse is inside its borders. Try moving the mouse and clicking sh-Right several more times. sh-Right summons the System menu in all contexts in Genera. This is important to remember.

2. Click sh-Right again and now take a more careful look at the information the System menu is providing.

   The operations that the System menu provides to you are divided into three categories:

   Windows
   This Window
   Programs

   Windows refers to general window operations, as indicated by the items in that column (See Figure 138):

   - Create - Create a new window.
   - Select - Select one of the windows already established by Genera or created by you. Another pop-up menu appears, offering all the windows currently available in Genera.
   - Split Screen - Divide your screen so that two windows are completely visible at the same time.
   - Layouts - Operate on the geography of the screen display by restoring the previous state or saving the current state. You can do what Layouts does with Edit Screen and the Undo option.
   - Edit Screen - Modify the geography of the screen.
   - Set Mouse Screen - Set the screen the mouse is on if you have more than one screen (for example, a color console and a standard one).

   This Window refers to operations of the current window, that is, the one that you were looking at and typing to when you clicked sh-Right.

   - Move - Change the location of the current window on the screen.
   - Shape - Modify the shape of the current window (square to rectangle, and so on).
   - Expand - Make the current window larger by occupying unused space on the screen.
   - Hardcopy - Send an image of the current window to a printer. This is the same as pressing FUNCTION Q. See the section "FUNCTION Key".
   - Refresh - Redisplay the current window. This is the same as pressing FUNCTION REFRESH. See the section "FUNCTION Key".

- Bury - Shuffle the current window down to the bottom of the stack of windows, selecting the previous (next down in the stack) window as current.

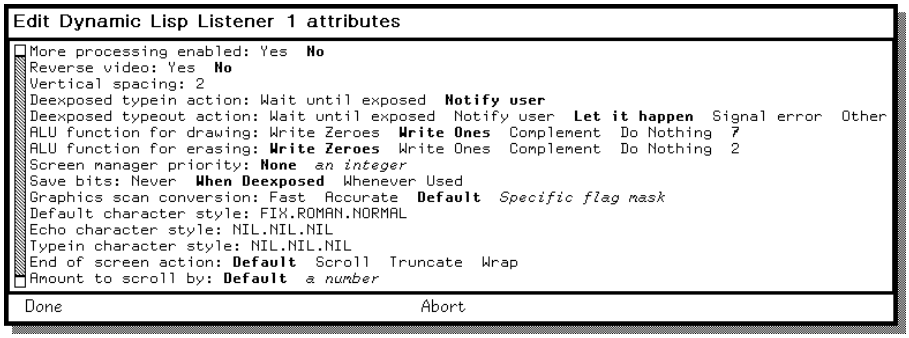
- Kill - Remove the current window.

- Reset - Initialize the current window, that is, restart it in its initial state. This is useful if a window is in a confused state. It might cause loss of information, however. See the section "Recovering From Errors and Stuck States".

- Arrest - Halt any processes running in the current window.

- Un-Arrest - Release any processes in the current window, allowing them to continue.

- Attributes - Modify the attributes of the current window. This summons a menu of the attributes of the window and allows you to selectively change them. Move the mouse down vertically and click on Attributes. A menu like this appears:



  Notice that the menu has two choices in its bottom margin: Abort and Done. These are for exiting from the menu. You can click on unhighlighted entries in the central portion of the menu (the choice section). Highlighted items are the current attributes. Clicking on an unhighlighted entry highlights it and selects it to go into effect, but nothing happens until you exit from the menu. Try clicking on several choices to see how it works. This is called an *Accept Variable Values* menu. Now, since you do not really want to modify anything just now, click on Abort. The menu goes away.

Click sh-Right again to get the System menu back and look at the third column:

Programs refers to available preloaded programs, or activities, in Genera.

You can select an activity from this menu, or you can use the Select Activity Command Processor command. See the section "Select Activity Command". For now, just move your mouse off the menu.

3. Click sh-Right once more. This time, move the mouse to the left and position it over the *Windows* column.

   Move it to sit on Create.

   Click Left on Create. Another momentary menu appears containing a list of window types.

   Click on Lisp.

   The menu disappears and an icon representing the upper left corner of a box appears where your mouse cursor was. Using the mouse, move this icon to some convenient location on your screen. (See Figure !).

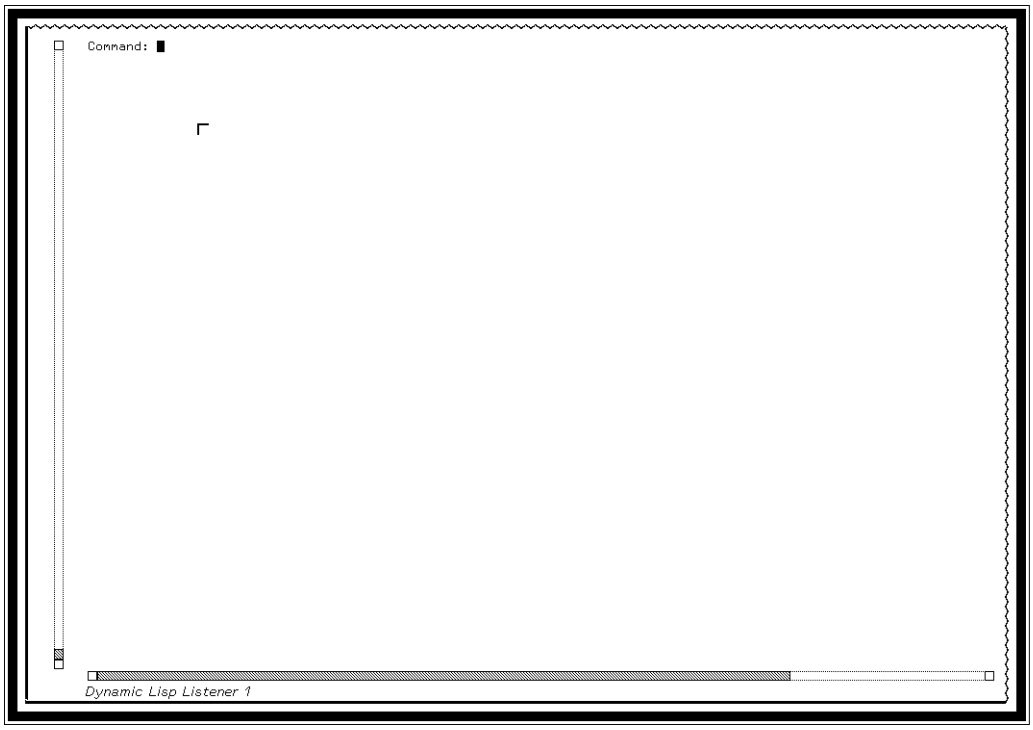


Figure 139. Positioning the Upper Left Corner of a New Window

Press the left mouse button and notice that the left corner now sprouts two more sides and becomes a *rubber band* with four corners (see Figure !). Stretch it around by moving the mouse. Notice what happens if you pull it up to the left of the left corner that you just positioned. If you were to click Left at this point, you would create a new window that is the full size of the screen.

Now, pull it down to make a reasonably sized window, at least ten inches across and at least 25 lines (about four inches) vertically. Click Left if your

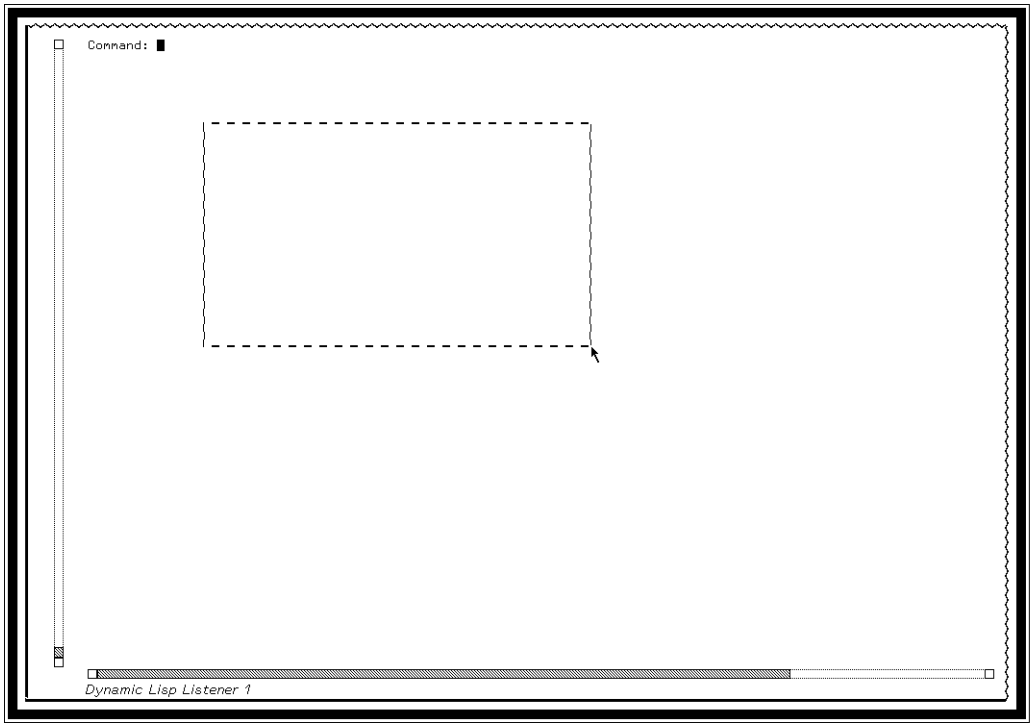


Figure 140. Positioning the Lower Right Corner of a New Window

mouse and the new window are not near any other window edge. If the lower right hand corner of your new window is near the edge of another window, click Right. Clicking Right (as the mouse documentation line says) is *smart*. This means it tries to optimize the position of the edges of windows so that they do not overlap. Windows that overlap by very tiny amounts at the edge can confuse you as to which window is actually selected, especially if you choose not to use the graying option for non-selected windows. (See the section "Set Screen Options Command".)

You have just created a second Lisp Listener, and it is selected. That means that any typing you do at this point goes into this window.

Trying Out the Command Processor

Typing Commands

Type `Select Activity Lisp` and press `RETURN`. You are now back in your original Lisp Listener. Type `Select Activity Lisp` and press `RETURN` again to get back to your newly created Lisp Listener. Select Activity *activity* cycles through all the windows of a given activity. Get back to your original Lisp Listener, typing `Select Activity` if necessary. Notice that there is a key labelled `SELECT`. Press `SELECT` fol-

lowed by L. You are now in your other Lisp Listener. SELECT L is the same as Select Activity Lisp. Press SELECT followed by HELP to see a display of the other single-letter abbreviations for activities.

Select Activity is a command that takes one argument, an activity name. Some commands take no arguments. Some take more than one argument. Some commands also take optional arguments called *keywords* that control the behavior of the command. For more detailed information about commands, see the section "Overview of the Command Processor".

Using the HELP Key

Type Select Activity and press SPACE, then press the HELP key. You should see a display like that in Figure !.

```
Activity (program) to interact with next
Type of input expected:  an activity

There are 23 possible activities.  Do you want to see them all? (Y or N) Yes.
These are the possible activities:
  Converse            FEP-Tape          Graphic Editor  Namespace Editor        Print Spooler          Terminal
  Distribute Systems  File ... (2)      Inspector       Notifications           Printer Spooler Log    Zmacs
  Document Examiner   Flavor Examiner   Lisp            Peek                    Restore Distribution   Zmail
  Editor              Frame-Up          Mail            Presentation Inspector  Select Key Selector

Command: :Select Activity (activity name)
```

Figure 141. Select Activity Command

The HELP key offers help appropriate to the situation. Press CLEAR INPUT. The command goes away. ABORT would also cancel the command, but since you had not finished typing the *arguments* to the command (that is, the activity you wanted to select), there is nothing to abort.

Try pressing m-HELP. The information displayed is called a *HackSaw*. HackSaws are helpful hints about Genera. See the section "HackSaws".

Using Keywords Arguments

Type Show Herald and press RETURN. The Herald (the initial screen display announcing the system version loaded) is printed on your screen. Show Herald is a command that takes no arguments.

Type Show Herald again and then press SPACE. The prompt (keywords) pops into your input line. Press HELP. You should see:
This time add :detailed before pressing RETURN. Now you should see additional information about the version of the FEP you are running with and other systems that are loaded in your world.

This demonstrates in a very simply way how keywords can affect the behavior of a command.

```
You are being asked to enter a keyword argument

These are the possible keyword arguments:
:Detailed             Whether to print version information in full detail
:Output Destination   Redirects the output of this command to the specified streams.

Command: Show Herald (keywords)
```

Figure 142. Show Herald and its Keywords

Some Useful Command Processor Commands

1. Type `Show Machine Configuration`. With no arguments, this command displays the information about your machine. You can give Show Machine Configuration the name of another machine on your network as an argument and it shows you the information about that machine. This is an important command to know, because if you have to call Symbolics Software Support, you might be asked for information about your hardware that you can get only from this display. Of particular importance when you call Software Support is the Machine Serial Number, appearing in the third line of the display.

2. Type `Show Command Processor Status`. This command takes no arguments because it is displays information about the setting of the Command Processor. Unless you have changed the Command Processor Mode or your prompt, you should see a display like this:

```
The command processor's current mode is
 Command Preferred: Interprets input starting with an alphabetic character as commands;
                    type an initial , to force Lisp interpretation.

The prompt string is "Command: ".

The prompt strings for other modes are:
  Form Preferred:          ""
  Form Only:               ""
  Command Only:            "Command: "
```

   This means that your Command Processor is in the default mode, and any typing you do to it is assumed to be a Command Processor command unless it begins with a left parenthesis or a comma. Type

```
,*package* SPACE
```

   This asks for the value of the Lisp variable ***package***. The value returned should match the package shown in the status line at the bottom of the screen.

   You can change the mode of the Command Processor. See the section "Set Command Processor Command".

3. Type `Show FEP Directory` and press `RETURN`. This displays a list of all the files on the local disk(s) of your machine. The files that Genera is currently using (the *world load* from which Genera was booted and the paging files loaded, and, for 3600-family machines, the microcode in use) are displayed in **boldface**. That is to remind you that you should not delete them.

Type Show FEP Directory again and this time press SPACE. When it prompts for keywords, press HELP. Try Show FEP Directory with the :type keyword. After typing :type press HELP again and select a kind of file, perhaps boot. This displays only those files that are boot files, that is, files that contain a script of commands for booting a world. See the section "Cold Booting After Powering Up".

Looking Back Over Your Output (Scrolling)

By now you have had to press SPACE at **More** breaks several times. Hold down the META key and press SCROLL (this is usually denoted m-SCROLL). You can look back over your interaction with Genera. To get back to the "end" of this *output history* you can press SCROLL, press m-sh->, or just start typing a new command. Without pressing anything, from where you are in the middle, try typing Show Namespace Object user *your user name*. If your user name were KJones, you would see the window scroll forward to the new command line and then display:

```
View in namespace ACME:
USER KJONES
LISPM-NAME KJones
PERSONAL-NAME "Jones, Kingsley"
HOME-HOST ACME
MAIL-ADDRESS kjones ACME
LOGIN-NAME kjones VAX01
NICKNAME King
WORK-ADDRESS "Building 3-701"
WORK-PHONE 5891
BIRTHDAY "19 June"
PROJECT Database
SUPERVISOR "Augensen"
USER-PROPERTY :USUAL-LOGIN-HOST wombat
```

For more details about scrolling windows, see the section "Scrolling with the Mouse".

Now you have tried a few Command Processor commands and it is time to show you how can speed up your work by cutting down on your typing. Go on to the section "Getting Acquainted with Dynamic Windows".

Getting Acquainted with Dynamic Windows

Try these things in a Lisp Listener (press SELECT L if you are in some other activity).

Reusing Commands

1. Press c-m-Y.

Press m-Y.

Press m-Y again.

Notice how each successive previous Command Processor command you typed is placed at the Command: prompt. Conveniently, you can reactivate any of these commands by pressing RETURN when the one you want appears. For now just press CLEAR INPUT.

2. Type Show Documentation SPACE Show Documentation Command RETURN. The documentation for the Show Documentation command is displayed on your screen. (You can use the Show Documentation command on any topic in the documentation set. See the section "Document Examiner".)

3. Press c-m-Y. Press m-RUBOUT three times to erase "Show Documentation Command". Now type "Reusing Commands" RETURN. You can yank back a command and edit it instead of retyping it.

4. Press ESCAPE. The 20 or so most recent commands in your *input history* are displayed, followed by (*n* more items in history), if you have typed more than about 20 commands. c-*n* ESCAPE displays the *n* most recent commands. Select a command and press c-*n* c-m-Y where *n* is the number of that command. Command *n* is yanked back at the Command: prompt.

Go on to the section "Using Your Output History" to learn more ways to take advantage of your output history.

Using Your Output History

1. Type Show Directory SPACE sys:examples;. This is a directory of sample programs you can look at and run.

   Move your mouse slowly over the display of the directory. Notice that the individual files (or subdirectories) listed are mouse sensitive, that is, a box appears around them as the mouse passes over them.

2. Now type Show File and click the left mouse button on one of the files in the display from Show Directory, perhaps on Teach-Zmacs-Info.text. The pathname is inserted in your new command line. Press RETURN to activate the command, or CLEAR INPUT if you do not want to see the contents of the file displayed. (This file, Teach-Zmacs-Info.text, is a good one to remember when you are ready to learn about Genera's editor, Zmacs. It tells you how to run a tutorial that explains the editor.)

   Type Show File again, and this time position your mouse at the right hand side of the directory listing. Click and hold the left mouse button and drag the mouse down and to the left. All the pathnames in the rectangle you cre-

ate by the diagonal motion are added as a sequence to the Show File command when you release the mouse button.

Note that you can mark groups of objects using the click and hold action, but if your mouse is over an object of the same type as the group you want to select (that is, if your mouse is over a single pathname, in this case), that single object is selected. To select a group, your mouse must be over a part of the screen that is not mouse sensitive.

3. Scroll back, using m-SCROLL or m-V, over your output history so far. Select a command line you would like to reactivate. Move the mouse over it and when it becomes mouse sensitive, click Left. The entire command line is *yanked* down to the current prompt. You can press RETURN to reactivate it, or RUBOUT or other editing commands to edit it. See the section "Editing Your Input".

4. Once again scroll back over your history and select a command to reactivate. Click sh-Left, that is, hold down the SHIFT key while you click the left mouse button. This time your selected command is not only yanked down to the current prompt but also is reactivated without your having to press RETURN.

5. When your history is long, scrolling back over it is tedious. Hold down the SUPER key and press R (s-R). Now type a string of characters, Show for instance. Notice that your cursor is moved back through to the history to the most recent occurrence of the word show. Press s-R again. You are moved back to the next most recent occurrence. If you move past the occurrence of show that you want, press s-S to search forward. Press END to terminate the search. You can now click Left or sh-Left.

6. Sometimes you know what command you want to yank and do not need to search for it. Hold down the CONTROL, META, and SHIFT keys and press Y (this is denoted c-m-sh-Y). You are prompted for a character string to use; the most recent command that contains that string is yanked directly. If the most recent command containing the string is not the one you want, press m-sh-Y and the previous most recent command is yanked instead. Successive uses of m-sh-Y go back farther and farther in your history locating commands containing the string.

Now that you know how to move back through your history it is time to learn some more ways to make use of it. Read on to the section "Using the Mouse".

Using the Mouse

1. Move your mouse so it is just off the top of the screen. Look down at the *mouse documentation lines* just above the date and time at the bottom of the screen. Notice that they are blank. Move the mouse back down onto a blank part of the screen. Now the mouse documentation line says "To see other commands, press Shift, Control, Meta-Shift, or Super".

2. Move the mouse so that it is over a previous command. Now notice the mouse documentation line. It now says that Mouse-L is the command, meaning that clicking Left yanks the command. You have already discovered this, see the section "Using Your Output History". Mouse-R says "Menu". Click Right. A menu pops up with a list of operations that can be performed on the command line the mouse is over. Move your mouse off the menu to make it disappear.

3. Now position your mouse over the command line again. Press SHIFT. Notice that the documentation changes to indicate what operations can be performed by pressing SHIFT (sh-) while clicking Left or Right. You have already tried sh-Left; see the section "Using Your Output History". sh-Right pops up the System menu, see the section "Using the System Menu".

4. Release SHIFT and press CONTROL. The operations offered by pressing CONTROL while clicking the mouse (c-) are for marking regions or words for yanking or copying. c-Left marks a region. Hold down the CONTROL key and the left mouse button while you move your mouse around. Notice that the text the mouse moves over is underlined. Release the CONTROL key and the left mouse button. Now click c-Right. The *Marking and Yanking* menu pops up, offering you several things to do with the text you have marked. Usually you want to push the text on the *kill ring*, the place where text that has been deleted with one of the kill commands (c-K, c-W, and others) in any context (Lisp, Zmacs, or Zmail) is stored for recall with c-Y. You can push the marked text on the kill ring by clicking on that choice in the menu. Do that now. Because pushing text on the kill ring is such a common operation, it can also be done by holding down SUPER and pressing W (s-W). Now press c-Y. The text you marked is yanked back at the current prompt. In this way you can yank arbitrary pieces of text for editing and turning into a new command. Since the kill ring is common to Lisp and Zmacs, this is a way to transfer something from your Lisp Listener to an editor buffer for editing or for saving in a file.

5. c-Middle marks and yanks the word the mouse is over. Try pressing c-Middle several times to yank arbitrary words from your output history. Press CLEAR INPUT.

6. Hold down both the CONTROL key and the SHIFT. c-sh-Middle allows you to mark (without yanking) words.

7. Holding down the META and SHIFT keys and clicking right pops up the *Window Operation* menu. This menu offers operations to perform on the current window, much like the "This Window" column in the System menu. See the section "Using the System Menu".

Using Menus

Using the Mouse and the Keyboard on Menus

Type `Set Window Options` and press `RETURN`. An Accept Variable Values menu is displayed. The items in the menu are the various options that you can set to customize your Lisp Listener. The highlighted (boldface) items are the current settings. Move your mouse over the menu and notice that items become mouse sensitive. You can click on an unhighlighted choice in a list to make it highlighted. You can click Left on a displayed value to replace it with a new value. You can click Middle on displayed value to edit the value.

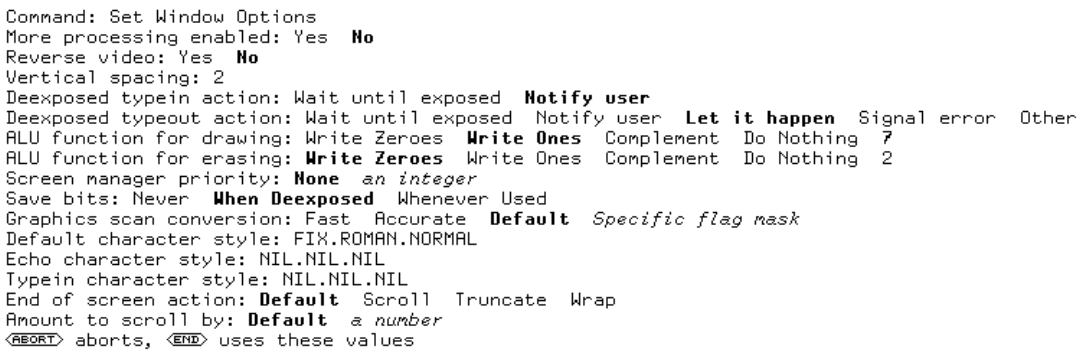


Figure 143. Set Window Options Menu

You can click on END to use these values, or on ABORT to cancel the command. You can also use keyboard commands to interact with this kind of menu. The keyboard commands available are:

SPACE	Enter a value for an item.
c-E	Edit the value of an item.
REFRESH	Force complete redisplay.
HELP	Display this list of commands.
END	Use these choices.
ABORT	Abort these choices.
c-N	Move down to next item.
c-P	Move up to previous item.
c-F	Move to next choice in a list of choices.
c-B	Move to previous choice in a list of choices.

Press `c-N` and notice that the first option, "More processing enabled" is underlined. Press `c-F`. The underlining moves over to the word **Yes**. Press `c-F` again. Now the underlining is under the work "No". Press `SPACE`. The line redisplays with the **No** choice in boldface. Press `c-N` to move to the next option, "Reverse video". Press `c-F`. Suppose at this point you decide you do not want to change the video after all. Just press `c-P` (or `c-N`) to continue. Press `c-P` now to return to the "More processing enabled" and `c-F` followed by `SPACE` to set more processing back on. Now press `END` to exit from the menu.

Occasionally typing on a menu causes some overwriting of other parts of a menu. Pressing REFRESH or FUNCTION REFRESH redisplays it correctly.

Using m-COMPLETE

Type Show Directory sys:examples;. The Show Directory command takes several keywords to control the format of the display. You can type them directly in the command line, but with some commands the interactions among keywords is complex. It is more convenient to see all the options and be able to alter them selectively. Press m-COMPLETE. You should see a menu like Figure !.

```
Files: SYS:EXAMPLES;*.*.NEWEST
Size: 0
Since: a universal time in the past or a null value
Before: a universal time or a null value
Order: Smallest-First  Largest-First  Oldest-First  Newest-First  Name  Type
Output-Destination: a destination
<ABORT> aborts, <END> uses these values
```

Figure 144. Show Directory Command Menu

The items in the menu are mouse sensitive; you can select keyword values with the mouse or by keyboard commands. Pressing END or clicking on <END> uses these activates the command.

Summary of User Interface Features of Genera

1. Scrollable History — SCROLL, m-SCROLL, c-V, and m-V scroll the history forward and backward.

2. s-SCROLL scrolls the window horizontally.

3. Click Left — on a directory listing, does a Show File of the file. In Other contexts yanks the command line.

4. Click Left on multiple objects — type Delete File and click Left on several file names from a Show Directory listing. Notice that the commas are added automatically.

5. Click sh-Left — like Left, but also activates. Click sh-Left on a command line to yank and activate the command.

6. c-Left — marks a region, m-W pushes marked region on the kill ring.

7. Click Middle — on a Lisp object does a Describe of the object.

8. c-Middle — yanks the word the mouse is over. Useful for using arbitrary text to compose commands; for example, after a Show Mail, click c-Middle on a pathname mentioned in a mail message as an argument to Show File.

9. Click Right — on an object, pops up a menu of possible operations on the object.

10. m-sh-Right — gets the menu of window operations.

11. Click Right — Press ABORT in the Dynamic Lisp Listener. Then click Right on the words Dynamic Lisp Listener 1 in the message

    ```
    Back to top level in Dynamic Lisp Listener 1
    ```

    Click on Show Flavor Components in the menu that appears.

12. c-R — search back through the output history. (c-S searches forward.)

13. c-m-sh-Y — prompts for a string and searches the typein history for a line matching that string and yanks it.

14. Non-trapping scroll bars. Clicking near the top offers scrolling by lines, near the bottom by pages, and the middle proportionally.

15. Addresses in messages (drafts being composed and messages received and the output of Show Expanded Mailing List) are mouse sensitive (see the section "Send Mail Command").

16. m-COMPLETE — pops up a menu of the command arguments for a command you are typing. You select values from the menu. END activates, ABORT aborts.

17. m-Left in Zmacs — Edit Definition. Hold down the left button and move the mouse around to see what is mouse sensitive.

18. m-Middle in Zmacs — Evaluate form. Hold down the middle button and move the mouse to see what is mouse sensitive.

19. SUSPEND — items on the window are still mouse sensitive (*cross window mouse sensitivity*).

20. Many menus accept keyboard commands for selection and activation. **dw:menu-choose** and AVV style menus can be typed at as well as clicked on with the mouse. These are the menus with grey shadows (as opposed to black shadows) and those that appear on your Dynamic Lisp Listener in the typeout window, such as that resulting from the Set Screen Options command or from using m-COMPLETE on a partially typed command. Press HELP for a list of the commands accepted. If the display becomes confused, press FUNCTION REFRESH.

What You Have Learned

If you have followed all the directions given in the sections starting with "Getting Acquainted with Genera", you should now be able to do the following:

- Use the System menu
- Use other menus
- Use the Command Processor to do some simple information gathering tasks
- Use the mouse
- Use facilities provided by Dynamic Windows to create new commands from your previous commands

You are now ready to learn in more detail about the command processor. See the section "Communicating with Genera".

For detailed descriptions of all the commands available in the command processor, see the section "Dictionary of Command Processor Commands".

To learn how to allow your application programs to take advantage of the power of Dynamic Windows and the Command Processor, see the section "Managing the Command Processor".

Communicating with Genera

Overview of the Command Processor

The Command Processor is a utility program that accepts a command and its arguments 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

The Command Processor operates in all Lisp Listeners and **break** loops. The prompt "Command: " indicates that you should enter a command or a Lisp form. By default, the Command Processor is in *command-preferred mode*. This means that input to a Lisp Listener or **break** loop is treated as a command if it begins with an alphabetic character or a colon. 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".

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

For descriptions of predefined commands: See the section "Dictionary of Command Processor Commands".

For information on the Command Processor reader and the facility for defining your own commands: See the section "Managing the Command Processor".

For information on turning the Command Processor on and off: See the section "Turning the Command Processor On and Off".

Parts of a Command

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

1. *Command name.* This is a word or a series of words separated by spaces. For example:

   ```
   Delete File
   ```

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 do not specify. For example, Show Directory takes one positional argument, a pathname:

   ```
   Delete File wombat:>KJones>program.lisp
   ```

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. These argument defaults are called *unmentioned defaults* and *mentioned defaults*. For example, Delete File takes several keyword arguments, :Expunge, :Output Destination, and :Query. The :Expunge keyword has values Yes and No. The unmentioned default is No. If you type:

   ```
   Delete File wombat:>KJones>program.lisp
   ```

   The file wombat:>KJones>program.lisp is marked for deletion but not expunged. The mentioned default for :Expunge is Yes, so you can type:

   ```
   Delete File wombat:>KJones>program.lisp :expunge
   ```

   which is the same as:

   ```
   Delete File wombat:>KJones>program.lisp :expunge yes
   ```

   In this case, the file wombat:>KJones>program.lisp is deleted and expunged.

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

For information on help in the Command Processor: See the section "Help in the Command Processor".

Entering Commands

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".)

The parts of the command can be entered using the keyboard or the mouse. You can click Left on previous commands on the screen to *yank* them for reactivation. If you click sh-Left on a previous command it is yanked and reactivated in one step.

When you type a command, items from your output history on the screen become mouse sensitive if they are appropriate as arguments to a command. Clicking Left on such an object yanks it into the current command line.

c-Middle yanks the word the mouse is over for use in composing a new command line. For example, if you have done Show Mail and a message refers to a file that you want to look at, you can yank the file name as an argument to a Show File command.

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. If you press m-COMPLETE the Command Processor displays a menu of the argument values it has collected so far. You can then select values from the menu using the keyboard or the mouse. 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:

```
de SPACE f SPACE foo.* SPACE :q SPACE y RETURN
```

You see the following on the screen:

```
Delete File (file [default WOMBAT:>KJones>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".

Supplying a Command Name

You type the command name, or some portion of it, followed by SPACE.

- 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:

  ```
  de SPACE f SPACE
  ```

  The Command Processor displays:

  ```
  Delete File (file [default WOMBAT:>KJones>foo.lisp])
  ```

- When it doesn't recognize what you have typed so far as the beginning of a command, the Command Processor informs you that no such command is 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.

- If there are no commands beginning with what you typed, you can use c-/ to see if there are any commands that contain what you typed anywhere in their names.

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:

```
se 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.

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 the arguments to the Delete File command and are now being prompted for keywords to modify the standard action of the command. You enter keywords and their values in any order, finishing 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:

```
de SPACE f SPACE foo.* SPACE :e SPACE n SPACE
:q SPACE a RETURN
```

The Command Processor displays:

```
Delete File (file [default WOMBAT:>KJones>foo.lisp]) foo.*
(keywords) :Expunge (Yes, No, or Ask [default Yes]) No
:Query (Yes, No, or Ask [default Yes]) Ask
```

You can also press m-COMPLETE to see a menu of the arguments and their values.

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.

Editing a Command

The Command Processor allows you to edit commands as you are entering them. You can move from field to field within a command, change arguments, delete keywords, even change the command name. The ability to edit input is feature of Genera called the Input Editor. A number of commands are available. See the section "Editing Your Input".

Help in the Command Processor

You can press the HELP key in 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-? or 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 argument, the Command Processor lists all the keyword arguments and briefly describes them. If you have started to type a keyword, the possible completions of what you have typed are displayed. If you have al-

ready typed one keyword and its value, only the unused keywords are displayed.

Keyword argument value
The Command Processor presents documentation for the meaning of all the possible values of the argument.

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.

Command History

Command Processor commands are maintained in the input editor history, along with other input to the Lisp Listener or **break** loop. c-m-Y yanks the last element of the history. m-Y yanks the next previous element. Thus you can press c-m-Y followed by m-Y m-Y ... to yank successively further back elements in your input 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.

c-m-sh-Y prompts you for a string and yanks the most recent element containing that string. m-sh-Y acts like m-Y, yanking successive previous elements.

Your output history is maintained on the Lisp Listener window. You can scroll back over your history using m-SCROLL or m-V. Scrolling forward is done with SCROLL and c-V, just as in Zmacs and Zmail.

c-R and s-R searche back through your history. END terminates the search so that you can yank the element you have found. c-S and s-S searche forward.

You can mark sections of your output to be pushed on the kill ring. c-Right pops up a menu of marking and yanking options, or you can mark elements directly using c-Left. Hold down the CONTROL key and the left mouse button and move the mouse over the area you want to mark. The marked region is underlined. You can push the marked region on the kill ring by clicking on that choice in the Marking and Yanking menu or by pressing m-W or s-W.

If you have two Lisp Listeners side by side on your screen, the histories of both remain mouse sensitive and you can yank elements from either one.

Clicking Right on an element of your history pops up a menu of possible operations on that object. For example, clicking right on a pathname offers, among other operations, a choice of Show File.

For a list of the mouse gestures that can be used on to manipulate your history on dynamic windows: See the section "Mouse Gestures on Dynamic Windows".

You can also copy your output history into a Zmacs buffer for editing or saving in a file. See the section "Copy Output History Command".

You can clear your output history if you want to clean up and do a garbage collection. See the section "Clear Output History Command".

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.

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". See the section "Setting the Command Processor Prompt".

For example, suppose you want the Command Processor to act as if it were not 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 **cp:*dispatch-mode*** to **:form-only** and **cp:*prompt*** to **nil** or the empty string. If you then want to return the Command Processor to its default behavior, you can set **cp:*dispatch-mode*** to **:command-preferred** and **cp:*prompt*** to "Command: ".

If for some reason you need to turn the Command Processor off completely, you can call **cp:cp-off**.

cp:cp-off *Function*

Turns off the Command Processor in all Lisp Listeners and **break** loops.

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

cp:cp-on &optional (*dispatch-mode* **cp:*dispatch-mode***) (*prompt-string* **nil** *prompt-supplied*) *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". This argument becomes the value of the variable **cp:*dispatch-mode***. The default mode is the current mode (the current value of **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 **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".

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.

Using the Zmacs Editor

Overview

Zmacs, the Genera 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 parts of the system are all based on Zwei, many of the commands available as Zmacs commands are available in other editing contexts as well. (In addition, many of the same editing commands are available in the Input Editor, which you use when typing commands or forms to other programs, such as the Lisp Listener. The Input Editor is not based on Zwei, however.)

Zmacs is used not only to create text for documents and programs, but also to compile programs, check them for correct structure, inspect parts of programs (including system programs), create commands, alphabetize lists, check spelling, and perform many other functions.

There is a tutorial on editing in Zmacs. For information on running it, type the following command:

```
Show File sys:examples;teach-zmacs-info.text
```

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.

You can create multiple copies of Zmacs by pressing SELECT c-E. SELECT E returns you to the last copy of the Zmacs process you used. Repeatedly pressing SELECT E cycles through all the copies of Zmacs.

For information on other methods of invoking Zmacs, see the section "Entering Zmacs".

Introduction to Inserting Text

Zmacs is always ready to accept an insertion. 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.

Commands

Zmacs *commands* are implemented by Lisp functions that perform the editing work. Every Zmacs command has a *name*, and many commands are bound to keys.

There are, in effect, three kinds of Zmacs commands, based roughly on how commonly used they are and how "serious" they are in their effects.

The first kind of Zmacs commands are the *keyboard accelerators*. Commonly used commands, such as Forward Word and Delete Forward, are bound to keys. Forward Word is on m-F and Delete Forward is on c-D. It would be tiresome to have to type a command each time you wanted to move forward a word or delete the next character. These commands also take *numeric arguments*. If you want to move forward three words, press m-3 m-F and if you want to delete the next fourteen characters, press c-14 c-D.

The second kind of Zmacs commands are the c-X commands. These commands take *two* keystrokes to invoke them, such a c-X c-S to save a file buffer, or c-X I to insert a file into the buffer, or c-X RUBOUT to kill the previous sentence. These commands also take numeric arguments where appropriate. You can see the entire list of c-X commands by pressing HELP C and then c-X followed by *. (There is also an interesting set of c-Q commands. You can see that list by pressing HELP C and then c-Q.)

Finally, there are the *extended* commands, commonly called the m-X (meta-x) commands. These commands are invoked by pressing m-X and then typing the command name and entering it. In general, these commands are more rarely used or have more long-lasting effects and are therefore slightly less easy to enter. Examples of these commands include Show Character Styles or Add Patch. In general, numeric arguments to these commands cause some unusual behavior, such as sending command output to a printer.

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.

Overview of Finding Out About Zmacs Commands

Sometimes you want to know if a Zmacs command exists that performs a certain function. Or, you might think that you know what a certain keystroke does, but you still want to make sure, or refresh your memory about its exact usage. This manual is one resource you might use in these circumstances. Zmacs itself has a number of built-in self-documentation facilities. This section describes some ways to get at this documentation.

Zmacs Command Completion

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 on the editor window or minibuffer to display a mouse-sensitive list of possible completions for the characters you have typed.

In addition, c-/ displays a mouse-sensitive list of every command that *contains* the substring and c-? displays a mouse-sensitive list of every command that *starts* with that string.

SPACE	Completes words up to the current word.
HELP or c-?	Displays possible completions in the typeout area.
Click Right	Pops up a menu of possible completions.
c-/	Displays a mouse-sensitive list of all commands *containing* the string you have typed so far.
c-?	Displays a mouse-sensitive list of all commands *starting with* the string you have typed so far.
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.

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".

Summary of Cursor Motion Commands

These are the Zmacs commands that you can use to move 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 — Up Real Line
Moves up one line.

m-] — Forward Paragraph
Moves to the start of the next paragraph.

m-[— Backward Paragraph
Moves to the start of the current (or last) paragraph.

c-X] — Next Page
Moves to the next page.

c-X [— Previous Page
Moves to the previous page.

c-V, SCROLL — Next Screen
Moves down to display the next screenful of text.

m-V, m-SCROLL — Previous Screen
Moves up to display the previous screenful of text.

m-< — Goto Beginning
Moves to the beginning of the buffer.

m-> — Goto End
Moves to the end of the buffer.

Getting Out of Trouble

Sometimes you type the wrong command. Mostly it is obvious what you have done wrong, and it is a simple matter to undo it. There are, however, some kinds of trouble you can get into that require special remedies. For example, you might accidentally delete large chunks of text you need or you might begin to type a command and then change your mind.

This section tells you how to recover from these situations.

Undoing

The Zwei Undo facility remembers all the changes that you have made in an editor buffer and allows you to selectively undo any or all of the changes you have made. The Undo facility is available from Zmacs, Converse, the Zmail draft editor, and other editor buffers based on the Zwei substrate. (It is not available from the Input Editor or in the minibuffer.)

The simplest operation of the Undo facility is to undo the most recent change to the editor buffer. Go to a buffer, type something in, delete it, and then press c-sh-U. The deletion is undone. Region marking shows what was undone. Now press c-sh-R. You're back where you started. It is always safe to undo, because you can always redo, and vice versa.

The Undo (m-X) and Redo (m-X) commands are similar to c-sh-U and c-sh-R with the added feature that a display in the minibuffer shows you what will be undone or redone before any action is taken. HELP U also displays the change before undoing it.

Keep pressing c-sh-U. Previous changes to the buffer are undone. You can keep doing this until the buffer is returned to its original state. When you reach this point, if the buffer contains a file, it's no longer marked as needing to be saved. And, if you undo all the changes to a section since it was compiled, it is no longer marked as needing to be compiled.

Repeated pressing of c-sh-R will successively restore the buffer until all the undo commands have been cancelled out.

If you read in a file with no intention of changing it and accidentally type some characters into it, use c-sh-U rather than RUBOUT to get rid of them. That way, the buffer is no longer considered to be modified.

Undo commands operate only on the current buffer. Each buffer has an undo history, and a separate redo history. The undo history can be displayed with c-0 c-sh-U. Likewise, the redo history can be displayed with c-0 c-sh-R. Items in the history are mouse-sensitive. You can undo or redo all changes *up through a given change* or you can undo or redo *any single change* in the history. By default, both histories are discarded when you save the buffer. See the section "Discard Change History".

Of course, subsequent changes may depend on the single change that you are undoing or redoing, so no guarantee can be made that undoing change number 13 in a 29-change history will have no effect on changes 14 through 29. (On the other hand, you can always back out of any undo or redo.)

This sounds more complicated in writing than it is when you are doing it. A few minutes experimentation in an editor buffer will make you a competent and confident user of the most important and common undoing and redoing operations.

After an undo or redo, the text that was modified is highlighted the same as if you had marked a region, but in this case there is no region, and the highlighting disappears when you type the next command. The history also shows you what constitutes each change. See the section "What is a Change to the Undo Facility?".

Large Deletions

Do not delete large pieces of text by repeatedly pressing RUBOUT and c-D. Apart from being slow, text deleted character-by-character is gone for good.

Instead, use delete and kill commands that save deleted regions in the kill history. c-K, m-K, and the commands that deal with *regions* easily wipe out and save larger chunks. Also, RUBOUT or c-D with a numeric argument erases that many characters all at once and saves them in the kill history. For full descriptions of these delete and kill commands, see the section "Deleting and Transposing Text in Zmacs".

Getting Text Back

The system has different histories for different contexts. One of these is always the *current history*. The two histories that you need to use for yanking in Zmacs are the *kill history* and the *command history*. The kill history remembers pieces of text that you killed or copied into it. In the context of Zmacs, the command history remembers all the editor commands that use the minibuffer in any way.

Additions to the histories are placed at the top of the list, so that history elements are stored in reverse chronological order — the newer elements at the top of the history, the older elements toward the bottom. A history remembers everything that has been typed to it since the last cold boot — it has no size limit.

Yanking commands pull in the elements of the history. *Top-level commands* start a yanking sequence; for example, c-Y yanks back the last text killed from the kill history, and c-m-Y yanks back the last command performed in the minibuffer. m-Y performs all subsequent yanks in the same sequence; for example, pressing m-Y while the kill history is the current history yanks the next item from that history.

A yanking sequence ends when you type new text, execute a form or command, or start another yanking sequence.

For complete descriptions of killing and yanking, see the section "Working with Regions in Zmacs".

Creating a Buffer

Zmacs creates your initial buffer when you first enter the editor. To create other buffers, use c-X c-F (Find File) to create either an empty buffer or a buffer containing a file. 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*
Genera	bork.lisp
UNIX	bork.lisp
VAX/VMS	bork.lsp

For more information about c-X c-F, see the section "Editing Existing Files".

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 Common-Lisp. See the section "Buffer and File Attributes in Zmacs".

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 creates an empty buffer, gives the buffer the new name, and displays (New File) in the minibuffer.
- To create a new buffer containing an existing 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.

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.

You can also write the buffer out with c-X c-W, Write File, if you want to change the name of the file from whatever you specified originally. 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.

Sending and Receiving Messages and Mail

Using Zmail

Introduction

Zmail is a display-oriented mail system for Genera. Using Zmail, you can send and receive electronic mail, archive your mail in disk files, and operate on groups of messages selected according to very flexible criteria. "Using Zmail" is intended to give you a brief introduction to the basic features of Zmail. For a complete description of all Zmail's capabilities, see the section "Zmail".

Zmail messages are composed in editor buffers, so some familiarity with the Zmacs editor is helpful. (See the section "Zmacs".)

Starting up Zmail

Before starting up Zmail, be sure that you are logged in. If you need help logging in, see the section "Logging In".

To enter Zmail, do one of the following:

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

You get a display similar to Figure !, called the *top-level display*. Now you can send or read mail.

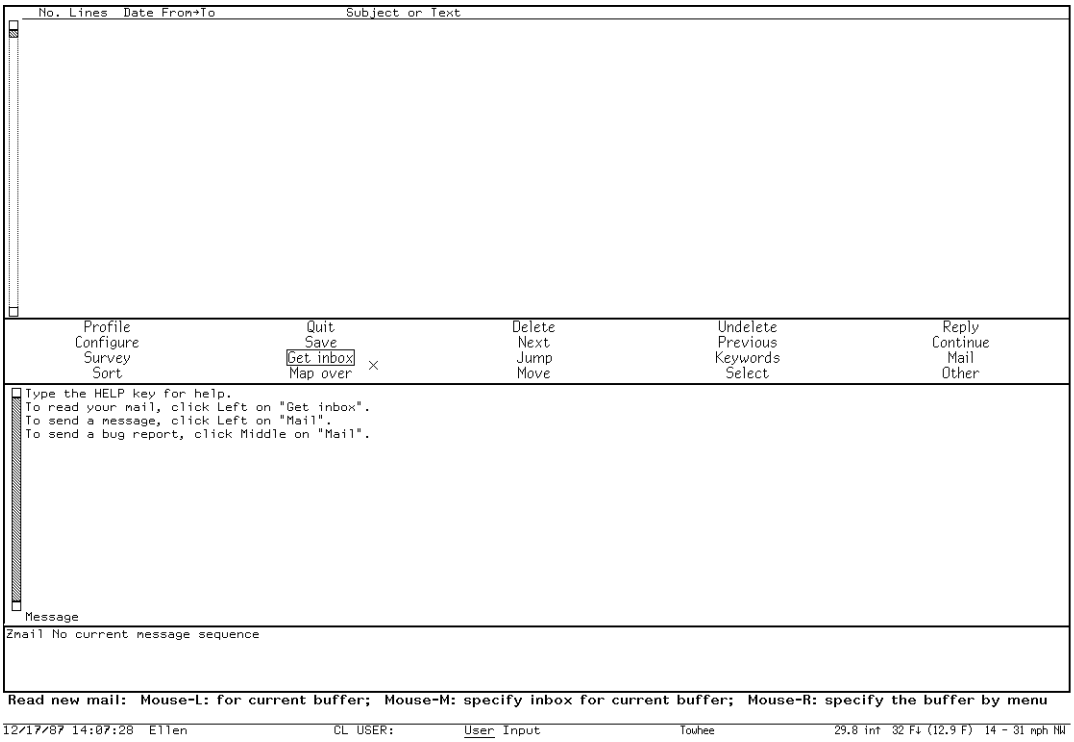


Figure 145. Top-level Display

The top-level display consists of four windows: the Summary Window, the Command Menu, the Message Window, and the Minibuffer, which contains the Mode Line.

The *Summary Window* displays a line for each message in the current sequence, with an arrow indicating the current message (see Figure 103).

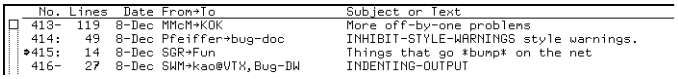


Figure 147. Zmail Summary Window

The *Command Menu* provides a mouse-sensitive menu of the most useful top-level commands (see Figure 104). In Zmail documentation, when we refer to "[Get inbox]", for example, we mean the Get inbox command in this menu. Some of these commands (for example, [Delete]) apply only to the current message.

Figure 148. Zmail Command Menu

The *Message Window* displays the current message (see Figure 105). The message window is an editor buffer.

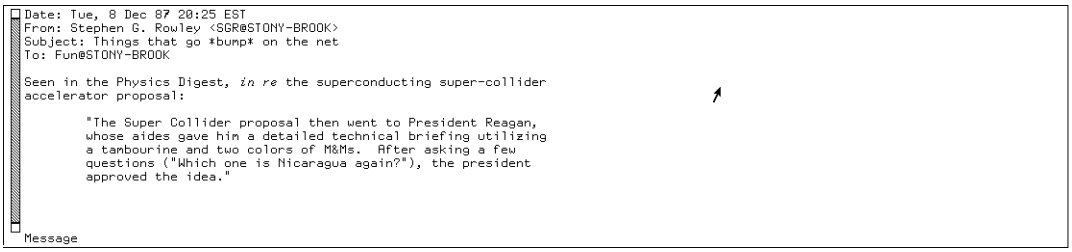


Figure 149. Zmail Message Window

The *Mode Line* gives status information about Zmail and about the current message, including its properties and keywords.

The top-level display, with a mail file read in, is shown in Figure !.

Command documentation is available online in several forms:

- Explanations displayed automatically; usually appear below the mode line.
- Mouse documentation line.
- HELP key: provides short command documentation.
- Apropos (M-X): lists commands whose name contains a given string.

See the section "Online Help for Zmail".

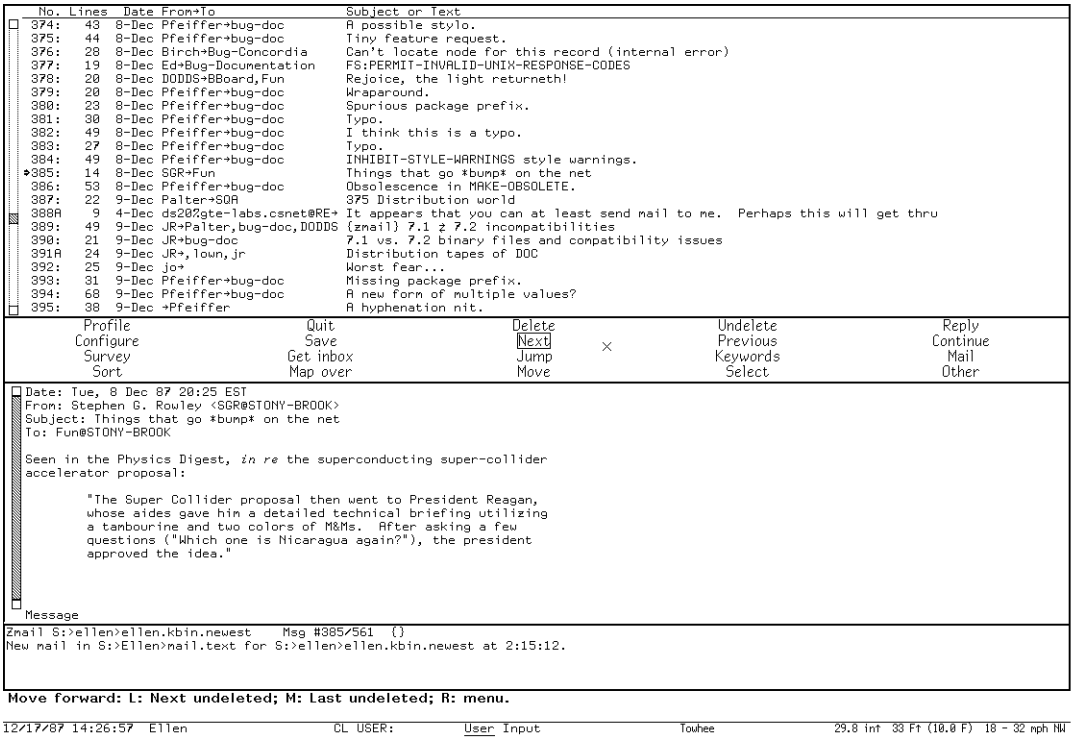


Figure 146. Top-level Display with Mail File

Sending Your Mail

To send a message, click on [Mail], which is displayed in the command menu.

[Mail] or `M` (Kbd)	Starts up a window for composing a mail message.
[Mail (M)]	Starts up a window for composing a bug report. You can control the behavior of clicking Middle in your profile. See the variable **zwei:*mail-middle-mode***.
[Mail (R)]	Calls up a menu of mail sending operations.

Zmail displays two windows, one for the message headers, and one for the message itself. See Figure !.

If you are sending a bug message, information about the software configuration of your machine is automatically added to the message window. (See Figure !.)

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.

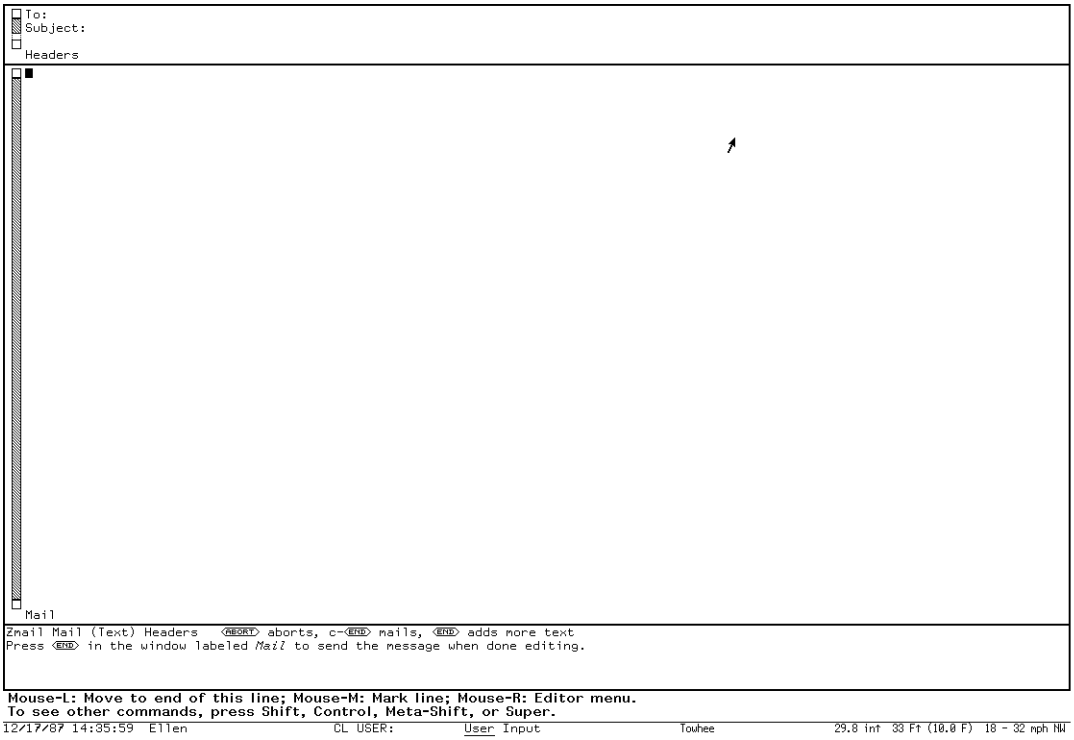


Figure 150. Mail Window

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).

Use c-N to get down to line containing the word Subject:. Fill in a short subject line for the message. This subject is 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 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". The headers window is also an editor window.

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. You can also change windows with c-X O, which puts you in the other 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

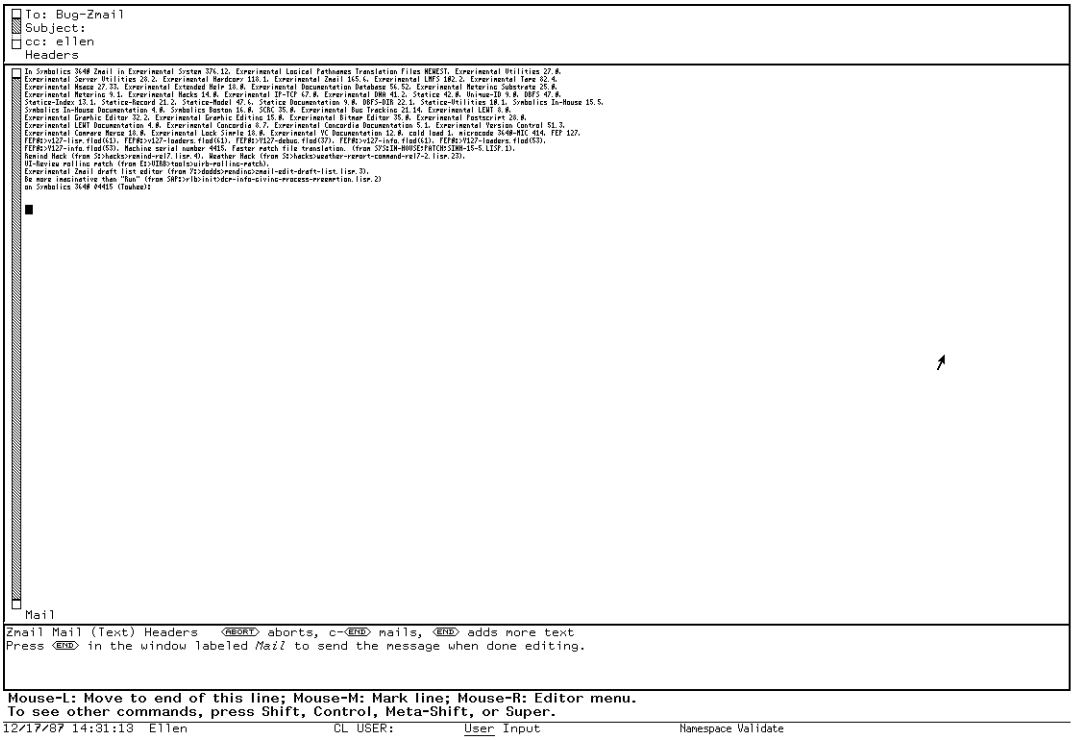


Figure 151. A Bug Mail Window

you later decide that you did want to send the message after all, use [Continue]. See the section "Continuing Completed or Aborted Zmail Messages".

When you are satisfied, press END to send the message. If you are in the headers window, press c-END. See figure ! 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 c-END.

Reading Your Mail

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

[Get inbox] or G from the keyboard

Gets the new mail (inbox) for the current buffer. If the current buffer is a collection (a group of related messages, see "Zmail Mail Collections") [Get inbox] has no effect.

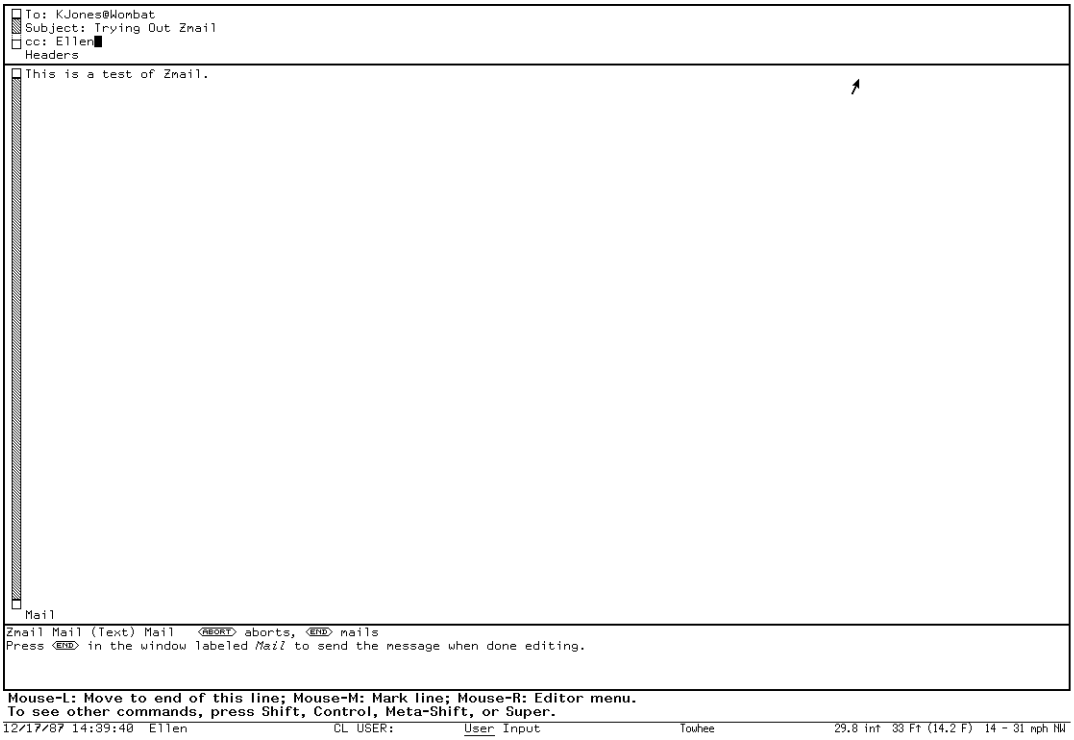


Figure 152. A Message about to be Sent

[Get inbox (M)] Prompts you for an inbox name for the current buffer.

[Get inbox (R)] Calls up a menu of possible buffers for which to get the new mail.

For a complete discussion of the [Get Inbox] command, see the section "[Get Inbox] Zmail Menu Item".

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 status line:

Parsing messages in *filename*

The parsing required in the creation of reference hash tables is time-consuming for large unparsed files. The appearance of this message notifies you that it is

building a reference hash table so that you do not think something is wrong. If you store your mail files in KBIN format, which is already parsed, this wait is eliminated. See the section "Binary Format for Storing Mail Files".

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 bottom edge of the window is a jagged line and the words `--more below--` appear in the mode line. When text is off-screen both above and below, both the top and the bottom edge of the window are jagged and the message reads `--more above and below--`; when you reach the final screen of the message, the top edge of the window is jagged and the message reads `--more above--`.

There are several ways to scroll using the keyboard:

To display the next screen of the message

```
SPACE
c-V
SCROLL
```

To go back to the previous screen

```
BACKSPACE
m-V
m-SCROLL
```

To return to the beginning of the current message

```
.
m-<
```

To use the mouse for scrolling, use the scroll bar in the left margin of the window. See the section "Scrolling".

What to Do After Reading a Message

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

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, click on [Delete] or press `D`. 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; click on [Undelete] or press `U`. This starts at the current message and searches backward for a deleted message, undeletes it, and selects it as the current message. When you delete a message from a mail buffer, the message is not actually removed — it just acquires

the property Deleted. You remove the message when you *expunge* the buffer; this happens automatically when you save it, or you can expunge it manually.

Moving Among Messages

When you finish reading a message that you do not want to delete, click Left on [Next] to read the next message. To go back to the previous message, click Left on [Previous]. To jump to the first message in the file, click [Previous (M)]; for the last message, click [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, c-m-V or c-m-sh-V.

Replying to Mail

To reply to the current message, click on [Reply].

[Reply] or R (Kbd)	Starts up a window to reply to the current message. You can customize the window configuration. See the variable **zwei:*reply-window-mode***.
[Reply (M)]	Starts up a window to reply to the current message with the message being replied to included. You can control the behavior of click middle in your profile. See the variable **zwei:*middle-reply-mode***.
[Reply (R)]	Calls up a menu of reply options.

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 !)

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 or c-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".

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 click Right or [Reply (R)] and choose from the pop-up menu the combination of To: and cc: you want. See the section "[Reply] Zmail Menu Item".

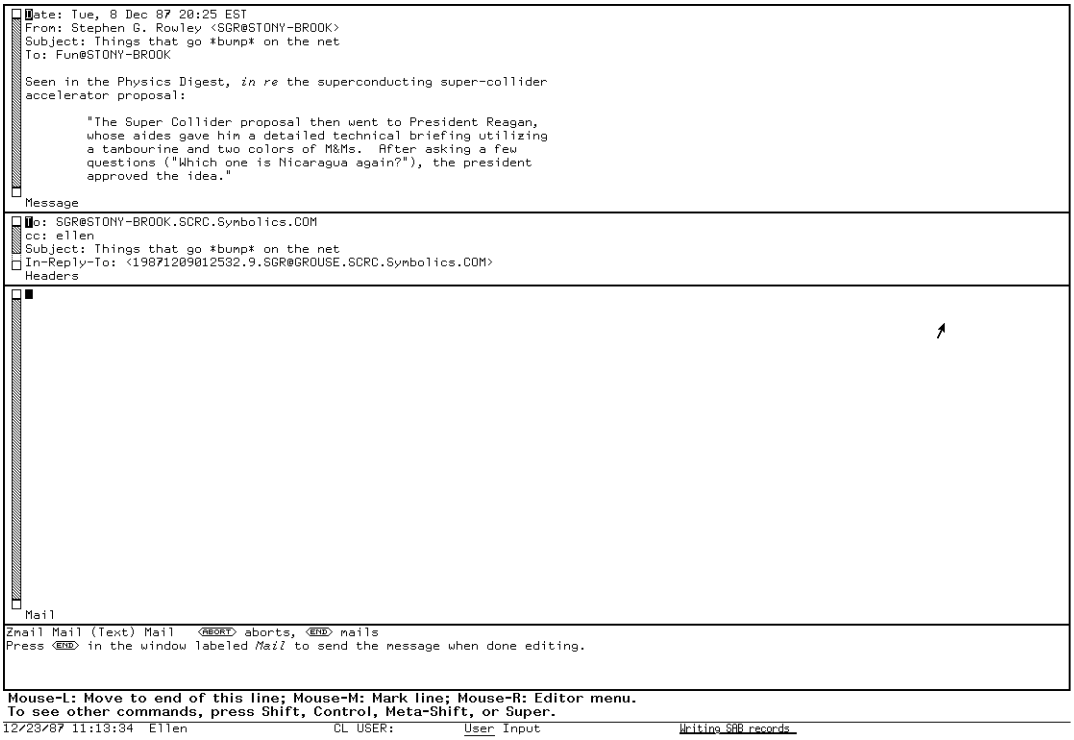


Figure 153. Mail Mode Display (Two-window Mode)

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.

Getting Fancy with Zmail

Once you have mastered the basics of Zmail, there are many advanced facilities that you can use for composing messages and for organizing your mail files. For example, you can take advantage of Genera's character styles to add emphasis to your messages; see the section "Using Character Styles in Zmail". This section touches on using keywords, one commonly used facility. For more detailed information and further suggestions, see the section "Zmail".

Zmail allows you to classify and categorize messages by adding keywords to them. Keywords are useful in many ways, among them:

Topic Indicators — Indicate the major topic of the message. If your work involves designing natural language interfaces, for example, you might use keywords such as `dictionary`, `parser`, and `syntax-checker`. The topic indicators you need depend on the sort of messages you get.

Classifiers — Indicate the type of message. For example, you might use keywords such as `bug`, `feature-request`, `documentation-bug`, and `issue` to categorize messages as bug reports, requests for features, reports of documentation bugs, and issues under discussion.

Status Flags — Indicate the status or priority of the message. For example, you might use a keyword such as `to-do` to flag messages that require you to do something and a keyword such as `timing-out` to flag messages on which you are awaiting action from other people. You could use `P1`, `P2`, and `P3` to indicate the priority of a message requiring further action.

To add keywords to the current message, click on [Keywords] in the Zmail menu. If you are using keywords for the first time, click right.

[Keywords (R)] — Pops up a highlighted menu of your keywords, in addition to the entry [New] for adding a new keyword. If you have never specified keywords for any messages, the menu contains only three items: [Do It], [Abort], and [New]. Click on [New] and type a keyword. The keyword appears on the menu, highlighted. Click on [Do It] and the keyword appears in braces on the summary line of the message. Keywords are stored in the mail files of the messages they are attached to. You can specify keyword/mail file associations explicitly in your Profile. See the section "Zmail Profile Options".

Figure 154. Keywords Menu

Clicking left on [Keywords] adds the last used keyword(s) to the current message.

You can sort your mail files by keywords, to have all the messages on one topic together. See the section "[Sort] Zmail Menu Item".

Your keywords appear in the menu offered by [Survey] so you can get a list of all the messages with a specific keyword attached to them. See the section "[Survey] Zmail Menu Item".

For more information about using keywords:

See the section "[Keywords] Zmail Menu Item".

See the section "Hints for Using Keywords, Mail Collections, and Mail Files".

Talking to Other Users

Introduction to Converse

Converse is a facility for communicating interactively with other logged-in users. A message sent with Converse pops up on the screen of the recipient almost instantaneously. The recipient has the choice of replying right in the pop-up window, entering Converse to reply, or doing nothing.

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, or pick text up and move it around between one message and another, or among messages, files, and pieces of mail.

To enter Converse, do one of the following:

- Press SELECT C.
- Give the Command Processor command Select Activity Converse.
- Evaluate **(zl:qsend)**.
- Click on [Select / Converse] in the System menu.
- Answer C in the Converse pop-up window when a message arrives.

Using Converse

Sending and Replying to Messages with 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: (see Figure 119).
You start a message by filling in a recipient after the To:, pressing RETURN and

Figure 155. A Fresh Converse Window

then typing the message text. It is not necessary to know what machine the person is using, but if you do know and give the recipient as *name@host* the message is sent considerably faster, since it is not necessary to search the namespace to find the machine (see Figure 120). To send the finished message, press END.
When the message has been sent successfully, it appears as part of 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 person to whom you sent the message filled in. Below this blank message, separated by a thin black line, the message you just sent appears, 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. (See Figure 157 .) Your cursor is left in the blank message so you can reply easily.

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

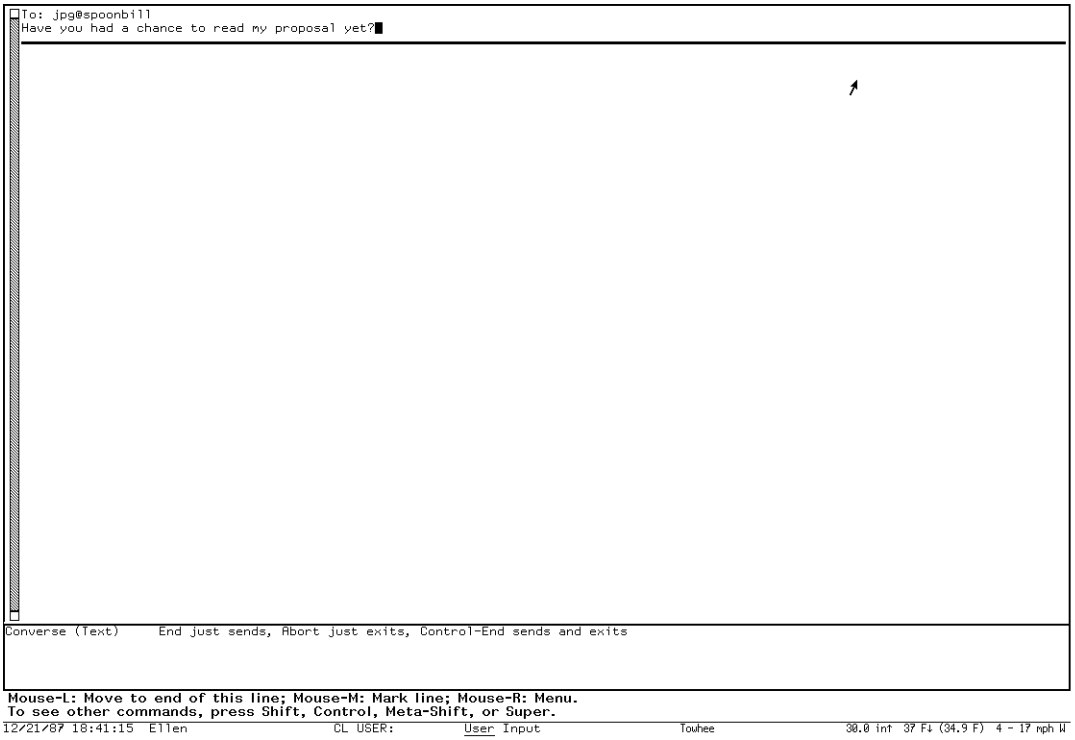


Figure 156. A Converse Message About to be Sent

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".

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

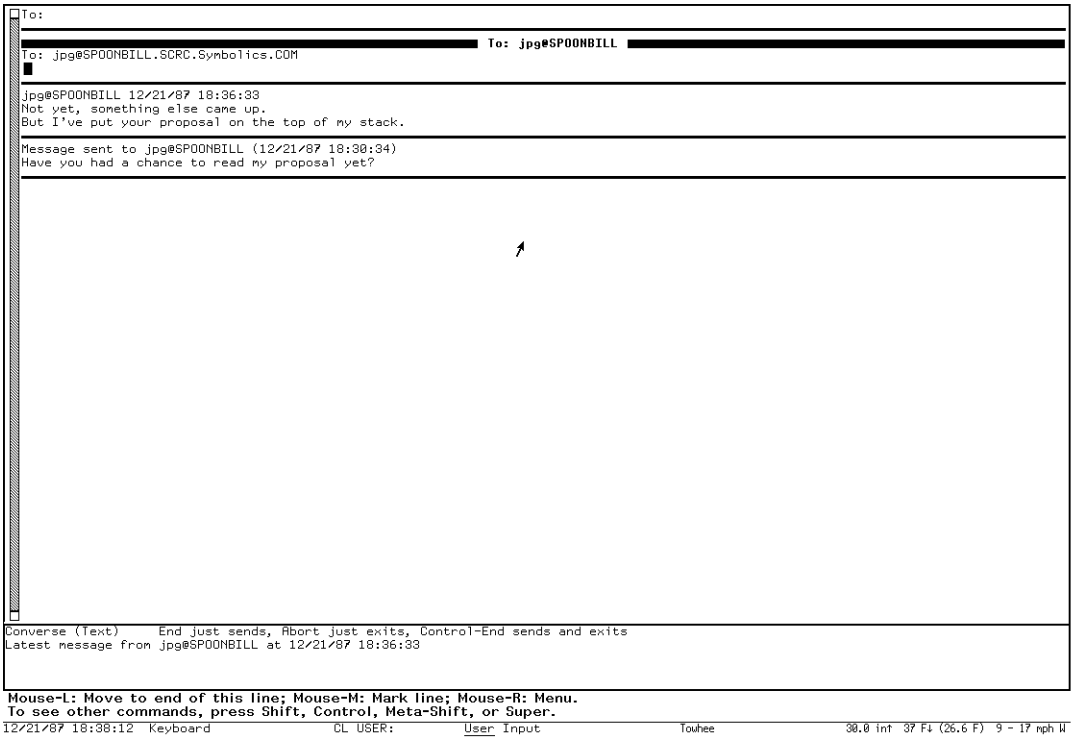


Figure 157. A Converse Conversation

enter Converse. Entering Converse has several advantages: you can look over the previous messages in the conversation, and you can use the Zwei editor to help you construct a reply. See Figure 122.

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.

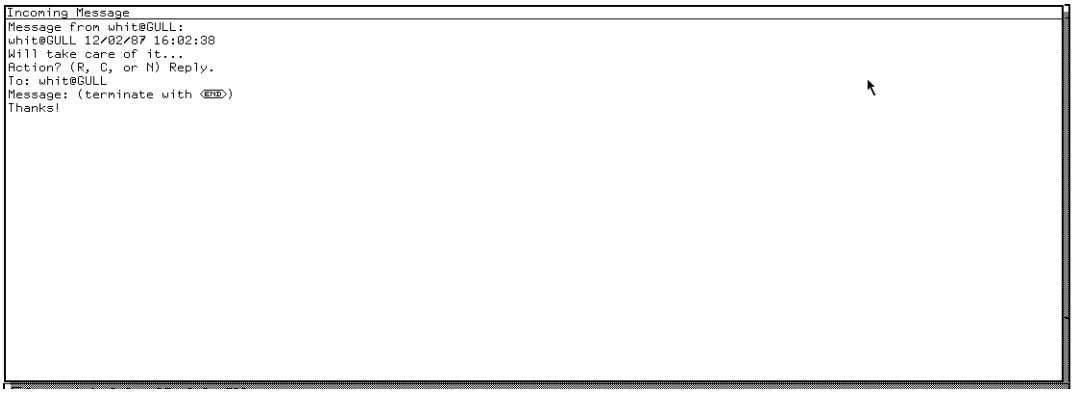


Figure 158. A Converse pop-up window

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 to whom you want to send the message is not logged in anywhere.
c-m-[	Moves to the previous conversation.
c-m-]	Moves to the next conversation.

Delete Conversation (m-X)
: Deletes the current conversation from the Converse window.

Write Buffer (m-X)
: Writes the entire Converse buffer (all conversations) to a file. It prompts for a pathname.

Write Conversation (m-X)
: Writes only the current conversation to a file. It prompts for a pathname.

Append Buffer (m-X)

Appends the entire Converse buffer (all conversations) to the end of a file. It prompts for a pathname.

Append Conversation (m-X)

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

Regenerate Buffer (m-X)

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 by marking it and using m-W before issuing the m-X Regenerate Buffer command.

Lisp Listener Commands for Converse

Command Processor Commands for Converse:

Send Message **Command**

Send Message *recipient*

Sends a Converse message to the specified recipient.

recipient — *user* or *user@host*. The person to whom to send the message. If *@host* is omitted, all Symbolics machines on your network are polled to locate *user*.

Send Message prompts for text to send as a Converse message. END terminates and sends the message. See the section "Converse".

Show Messages **Command**

Show Messages *keywords*

Displays the contents of the specified Converse conversations.

keywords — :Direction, :From, :Mention Empty Sequences, :More Processing, :Order, :Output Destination, :Query, :Recent, :Start, :Stop, :Summarize, :To

:Direction — {Incoming, Outgoing, All, or Default} Whether to show incoming messages, outgoing messages, or all. The Default is Incoming.

:From {*user-or-address*} Show messages from this user or address.

:Mention Empty Sequences
{Yes, No} Whether to mention empty message sequences, for example, you have sent messages to someone but the person did not reply. The default is No, not to mention this. If it is Yes, you see "No messages from *so-and-so*".

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Order {Forward, Reverse} How to order the message presentation within each conversation. The default is Forward, that is, most recent first.

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Query {Yes, No} Whether to ask about each conversation. The default is Yes, to ask.

:Recent {Yes, No} Whether to consider only the most recently exchanged messages in each conversation.

:Start {*number*} Number of first message to show in a conversation. If there are fewer then *number* messages in the conversation, that conversation is skipped.

:Stop {*number*} Number of last message to show in a conversation.

:Summarize {Yes, No} Whether to show the entire message or just a summary. The default is No, to show the entire message. If yes, messages are mentioned but not shown.

:To {*user-or-address*} Show messages to this user or address.

Lisp Functions to Control Converse:

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

Refuses interactive messages. 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** turns sends back on and clears **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.

net:notify-local-lispms &optional *message* &key *(:error-p) (:report) (:output-stream)* *Function*

Sends *message* to all Symbolics machines at your site based upon information it gets from the namespace database about the Symbolics 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.

Keyword arguments are:

:error-p
Setting this keyword to **t** enables the function to report all errors encountered. Specifying **nil** (default) for this keyword enables the function to ignore all errors encountered.

:report
Setting this keyword to **t** (default) enables the function to report whether it succeeded in delivering the message. Specifying **nil** enables the function to only report failures in delivering messages.

:output-stream
Using this keyword enables you to redirect output to a specific stream.

zl: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 give just a name, **zl:qsend** looks at all the Symbolics machines at your site to find any that *name* is logged into. If the user is logged in to one Symbolics machine, it is used as the host; if more than one, **zl:qsend** asks you which one you mean. If you leave out *destination* altogether, doing just **(zl:qsend)**, Converse is selected as if you had pressed `SELECT C`.

message should be a string. For example:

```
(qsend kjones@wombat "Want to go to lunch?")
```

If *message* is omitted, **zl: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 **zl:qsend**. So you get some editing power, although not as much as with full Converse (since the latter uses Zwei). See the section "Editing Your Input". **zl:qsend** predates Converse and is retained for compatibility.

print-sends &optional (*stream* **zl: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.

zl: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 **zl: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.

Customizing Genera

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 character style of the screen display, 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.

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:

LMFS	lispm-init.lisp
UNIX 4.1	lispm-init.l
UNIX 4.2, 4.3	lispm-init.lisp
VAX/VMS	lispmini.lsp
TOPS-20	lispm-init.lisp
ITS	*name* lispm

A simple init file consists primarily of the **login-forms** and the **zl:setf** 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 **zl:setf** special form sets the value of one or more variables.

Here is an example of a simple init file:

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10; -*-

(login-forms
  (zl:setf cp:*prompt* 'si:arrow-prompt)

  zwei:
  (zl:setf text-mode-hook 'auto-fill-if-appropriate)

  (zl:setf si:local-finger-location
        (cond ((y-or-n-p "in your office? ")
               "340 Domingo x562")
              (t (format t "~&Where are you? ")
                 (readline query-io)))))

(hardcopy:set-default-text-printer "Echo-Lake")
(hardcopy:set-default-bitmap-printer "Echo-Lake")
```

In this simple init file, the first **zl:setf** changes the value of the variable that displays the command processor prompt from the default `Command:` to an arrow. The second **zl:setf** specifies that the system automatically fill text that you type in any editor-based activity when appropriate. The third **zl:setf** 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.

How to Create an Init File

The easiest way to create an init file is by copying the sample init file 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 Genera longer than you.

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* *Function*

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
        (zl:setq-standard-value base 8)
        (zl: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.

Other variables can be set inside **login-forms** using **symbol-value-globally**:

symbol-value-globally *var* *Function*

Works like **symbol-value** but returns the global value of a special variable regardless of any bindings currently in effect (in the current stack group).

symbol-value-globally does not work on local (lexical) variables.

You can use **setf** with **symbol-value-globally** to bind the global value of a special variable. **(setf (symbol-value-globally** *var***))** ... **)** is the same as **zl:set-globally** and supersedes **zl:setq-globally**.

See the section "Functions Relating to the Value of a Symbol".

zl: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".

To load individual files from your init file, use the **load** function:

```
(load "SYS: LISP; MY-PROJECT")
(load "Tuna:>kjones>examples>decorate")
(load "vixen://usr//kjones//tools//toolkit")
```

The first sample form loads a file using its logical pathname; the second form loads a file from a LMFS using its physical pathname. The third form loads a file from a UNIX system in the appropriate syntax (the slashes are doubled).

If you want to put command processor commands in your init file, you can do so using the function **cp:execute-command**:

```
(cp:execute-command "show file" "foo.lisp")
(cp:execute-command "show herald" :detailed t)
(cp:execute-command "load system" "mysystem"
                    :compile :always :automatic-answer t)
```

cp:execute-command *command-name* &rest *command-arguments* *Function*

Invokes a Command Processor command from within a program. See also **cp:build-command**, which **cp:execute-command** makes use of.

command-name
: Symbol or string naming the command to invoke; if a string, it must be in the command table to which **cp:*command-table*** is currently bound.

command-arguments
: Positional and keyword arguments to the named command.

Examples:

```
(cp:execute-command "show file" "test-data.text")

(cp:execute-command 'si:com-load-system "unifier"
                :condition :always :automatic-answer t)
```

For an overview of **cp:execute-command** and related facilities, see the section "Managing the Command Processor".

Useful Customizations to Put in Your Init File

The number and kinds of customizations you can put in your init file is limited only by your imagination. This section offers some suggestions that many users have found useful, but it is by no means an exhaustive list. You might find some additional ideas for your init file among the HackSaws. See the section "HackSaws".

Adjusting Console Parameters

tv:set-screen-options &rest *vars-and-vals* &key *(:screen* **tv:main-screen***)* &allow-other-keys *Function*

Allows you to set the screen options under program control. **tv:set-screen-options** controls the same screen options as the Set Screen Options command menu, but since it is a Lisp function you can put it in your init file.

You can specify **:screen** to set options for a particular screen. The default for **:screen** is the value of **tv:main-screen**.

vars-and-vals are the same options as in Set Screen Options, used as keywords. Here are the keywords and their possible and default values, with the corresponding defaults in the Set Screen Options menu in parenthesis if they are not obvious.

Documentation line options:

:number-of-wholine-documentation-lines - 1 or 2. The default is 2.
:wholine-documentation-reverse-video-p - **t** or **nil**. The default is **t** (**reverse**).
:wholine-documentation-character-style - a character style appropriate for **:screen**. The default is **(:swiss :bold-condensed-caps :normal)**. See the section "Character Styles and the Lisp Listener".

Status line options:

:wholine-default-character-style - a character style appropriate for **:screen**. This keyword sets the variable **tv:*wholine-default-character-style***. The default is **(:fix :roman :normal)**. See the section "Character Styles and the Lisp Listener".
:wholine-clock-format - **:dow-hh-mm-ss** (Mon 31 Jan 11:59:59), **:month-day-year** (12/31/89 23:59:59) or **:dow-hh-mm-am** (Mon 31 Jan 11:59pm). The default is **:dow-hh-mm-ss**.
:wholine-clock-colon-blink-half-period - **nil**, a positive integer, interpreted as a number of seconds, or a string interpreted as a time interval, for example "2 minutes".
:show-current-process-in-wholine - **t** or **nil**. The default is **nil** (**user name**).
:show-machine-name-in-wholine - **t** or **nil**. The default is **nil** (**invisible**).
:wholine-file-state-character-style - a character style appropriate for **:screen** The default is **(:fix :extra-condensed :normal)**.
:note-progress-in-wholine - **t**, **nil** or **:with-file** (default **t**)

Global screen options: (These options are not applicable to MacIvory)

:beep-mode - **:beep**, **:flash**, or **t**. The default is **t** (**both**).

:dim-screen-after-n-minutes-idle - **nil**, a number of minutes, or a time interval. The default is 20 (**20 minutes**).

:screen-dimness-percent - an integer between 0 and 100. The default is 0.

Global window defaults:

:end-of-page-mode - **:scroll**, **:truncate**, or **:wrap**. The default is **:scroll**.
:scroll-factor - a number of lines or a fraction of the screen.
:context-nlines - a non-negative integer, the number of lines to overlap the screen when scrolling. The default is 1.
:noise-string-p (Whether to prompt in the CP) - **t** or **nil**. The default is **t**, meaning prompting within a command line is enabled. See the variable **si:*disable-noise-strings***.
:noise-string-style (Character style for prompts) - a character style appropriate for **:screen**. The default is **(nil nil nil)**, meaning the same as the default screen character style. See the section "Character Styles and the Lisp Listener".
:graphics-scan-conversion-mode (Graphics Scan Conversion) - a number or list of keywords and values.

Background interactor window settings:

:background-lisp-interactor-screen-utilization - Where to put the window when a background process wants to interrupt. The possibilities are **:at-top** or **:at-right.** The default is **:at-right.**

:background-lisp-interactor-screen-fraction - The fraction of the screen to cover. The possible values are from 1/3 to 1 (the full screen). The default is 3/4.

Keyboard and mouse repeat controls:

:keyboard-auto-repeat - **t** or **nil**. Is Auto-repeat enabled? The default is **nil.**

:enable-repeat-key - **t** or **nil**. Is the Repeat Key enabled? The default is **t.**

:keyboard-repeat-initial-delay - 60ths of a second before key starts repeating. The default is 42.

:keyboard-repetition-interval - 60ths of a second between repeated characters. The default is 2.

:scroll-bar-auto-repeat - (Scroll bar repeats when the mouse is held down) **t** or **nil**. The default is **t.**

:scroll-bar-repeat-initial-delay - 60ths of a second before scroll bar starts repeat. The default is 30.

:scroll-bar-repeat-minimum-lines-per-second - Minimum number of lines scrolled per second. The default is 2.

:scroll-bar-repeat-maximum-lines-per-second - Maximum number of lines scrolled per second. The default is 50.

:scroll-bar-repeat-maximum-screens-per-second - Maximum number of screens scrolled per second. The default is 2.

Other:

:console-who-line - whether the Status line is present on main screen. **t** or **nil**, the default is **t.**

Per Screen options - Options that are applicable to the screen or screens designated by **:screen**:

:background-gray - a gray name, one of the symbols in **tv:*gray-arrays***. The symbols in **tv:*gray-arrays*** are: **nil stipples:5.5%-gray stipples:6%-gray stipples:7%-gray stipples:8%-gray stipples:9%-gray stipples:10%-gray stipples:12%-gray stipples:hes-gray stipples:25%-gray stipples:33%-gray stipples:50%-gray stipples:75%-gray :black** and **:white**. You can also define your own gray. The default is **:white.**

:deexposed-gray - a gray name as for **:background-gray**. The default is **stipples:6%-gray**. (**6** percent)

:bow-mode (Screen background mode) - **t** or **nil**. The default is **t** (**white**). *(This option is not applicable to MacIvory)*

:menu-highlighting-style - **nil** or a character style appropriate for **:screen**. The default is **nil** (`Inverse video box`, that is, use the same character style with inverse video).

For example, to set your screen options to have the mouse documentation line in black on white, your machine name appear in the status line, your clock format showing A.M. or P.M. instead of seconds, a medium gray surrounding the screen, and your prompts in bold, you put a form like this in your init file:

```
(tv:set-screen-options
        :wholine-documentation-reverse-video-p nil
        :show-machine-name-in-wholine t
        :wholine-clock-format :dow-hh-mm-am
        :background-gray tv:50%-gray
        :noise-string-style '(nil :bold nil))
```

dw::set-default-scroll-bar-margin *new-margin* *Function*

Sets the position of the vertical scroll bar to *new-margin*.

The default position of the vertical scroll bar is on the left of the window. To set it to the right, you evaluate:

```
(dw::set-default-scroll-bar-margin :right)
```

si:*kbd-repeat-key-enabled-p* *Variable*

Controls whether or not the REPEAT key is enabled. The default is **t**. It can be set using **zl:setf**:

```
(setf si:*kbd-repeat-key-enabled-p* nil)
```

Setting **si:*kbd-repeat-key-enabled-p*** to **nil** turns off repeating using the REPEAT key.

si:*kbd-repetition-interval* *Variable*

Controls the speed of repetition of characters when the REPEAT key is held down, in sixtieths of a second. Its default is 2, which is a thirtieth of a second between repeated characters.

si:*kbd-auto-repeat-enabled-p* *Variable*

Controls whether or not keys repeat if held down (auto-repeat). The default is **nil**, meaning that holding keys down does not cause repetition. It can be set using **zl:setf**:

```
(setf si:*kbd-auto-repeat-enabled-p* t)
```

Setting **si:*kbd-auto-repeat-enabled-p*** to **t** turns on auto-repeat. You can set the length of time a key must be held down before it starts to repeat with **si:*kbd-auto-repeat-initial-delay***.

si:set-auto-repeat-p *key* &optional (*state* **t**) *Function*

Allows you to specify keys that should not auto-repeat even if auto-repeat is enabled. By default all keys can auto-repeat except for FUNCTION, SELECT, NETWORK, ABORT, SUSPEND, and RESUME. For example,

```
(si:set-auto-repeat-p #\Square nil)
```

turns off auto-repetition for the SQUARE key. You can make SQUARE auto-repeat again by setting it back to **t**.

si:*kbd-auto-repeat-initial-delay* *Variable*

Controls how long you must hold down a key before auto-repetition starts, in sixtieths of a second. The default is 42, which is between half and three-quarters of a second. You can adjust it using **zl:setf**.

tv:screen-brightness *main-screen-mixin* *Function*

Returns the brightness of the screen as a floating-point number between 0 and 1. **(tv:screen-brightness tv:main-screen)** may be set in your init file using **zl:setf** to adjust the screen brightness. Console hardware varies slightly so you must experiment to find the value that suits you best. One technique for doing this is to adjust the brightness using LOCAL-B and LOCAL-D until it is to your liking. Then use **(tv:screen-brightness tv:main-screen)** to find that value. For example:

```
(tv:screen-brightness tv:main-screen) => 0.43307087
```

Then in your init file you place the form

```
(setf (tv:screen-brightness tv:main-screen) 0.43307087)
```

Each time you log in with your init file the screen brightness is automatically set to that value.

tv:*dim-screen-after-n-minutes-idle* *Variable*

Controls the length of time a console must be idle before its screen dims. You can set this in your init file to adjust the length of time it takes the screen dimmer to activate. The default is 20 minutes. Setting it to **nil** disables the screen dimmer entirely.

tv:*screen-dimness-percent* *Variable*

Controls the brightness value of the screen when it is dimmed. You can set this in your init file to adjust the dimness of the screen. The default is 0, meaning black. 100 is bright. If you want a number that will leave the screen very dim but visible, the value will vary with your particular hardware. Experiment to find a good setting, starting with 50.

sys:console-volume &optional *(console* **sys:*console***) *Function*

Returns the current volume setting for the console, which is a number between 1.0 (loudest) and 0 (softest). The console volume can be changed with **zl:setf**, as in the example:

```
(setf (sys:console-volume) 0.5)
```

si:*disable-noise-strings* *Variable*

Controls whether the Command Processor prints prompts within command lines. The default is **nil**, meaning that prompting is enabled and Command Processor commands prompt for arguments. Setting it to **t** disables this prompting.

Customizing the SELECT Key

You can adjust the keys that are used with the SELECT key to select activities. SELECT = starts up the SELECT Key Selector activity.

The initial frame looks like Figure !.

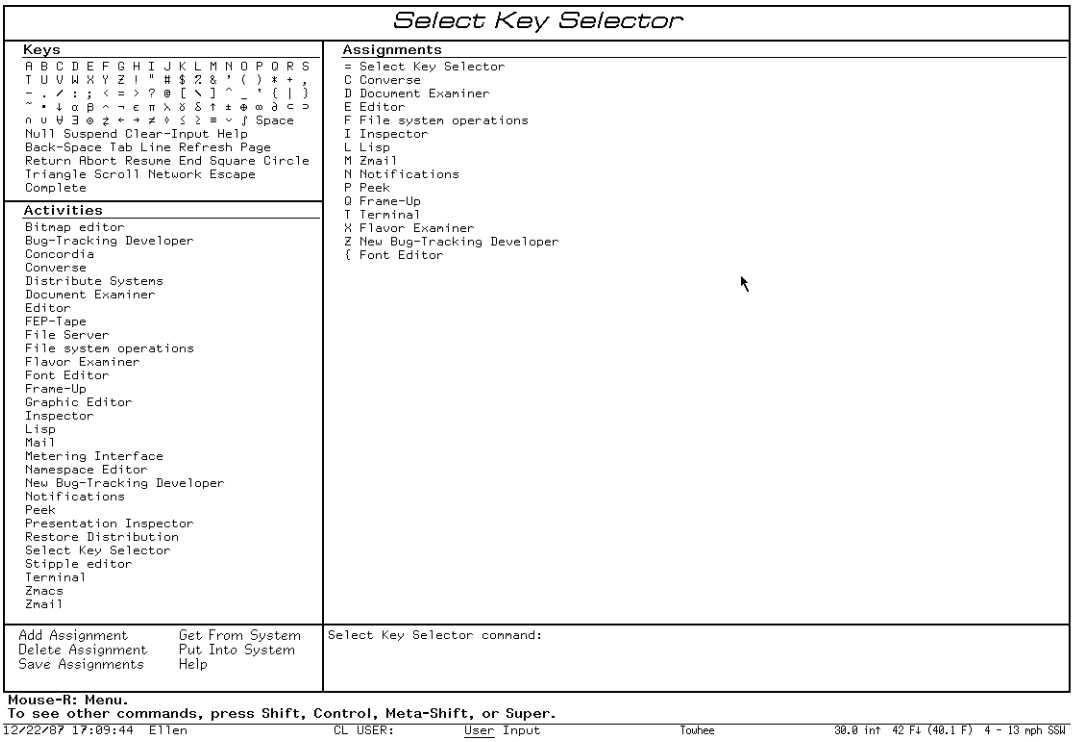


Figure 159. The SELECT Key Selector Window

Clicking Middle on the command names or activity names gives some documentation.

To add an activity to a SELECT key:

1. Click on [Add Assignment].
2. Click on or type the name of the activity you wish to assign to a key.
3. Click on or type a key to assign to that activity.
4. Click on [Put Into System].

To remove an activity from a SELECT key:

1. Click on [Delete Assignment].
2. Click on the name of the activity or the key you want to remove.
3. Click on [Put Into System].

You can save the assignments you prefer in your init file. Save Assignments constructs a Lisp form which represents your changes, and pushes that form onto your kill history. You can then yank the form into your lispm-init file. The form looks something like this:

```
(TV::SETUP-SELECT-KEYS
  '((#\= (DW::FIND-PROGRAM-WINDOW-FOR-SELECT-KEY
           'CL-USER::SELECT-KEY-SELECTOR TV:ALWAYS-MAKE-NEW)
    "Select Key Selector" T)
   (#\C ZWEI:CONVERSE-FRAME "Converse" T)
   (#\D (DW::FIND-PROGRAM-WINDOW-FOR-SELECT-KEY
          'DDEX::DOC-EX TV:ALWAYS-MAKE-NEW)
    "Document Examiner" T)
   (#\E ZWEI:ZMACS-FRAME "Editor" T)
   ...))
```

Starting up Zmail in the Background

You can start up Zmail from your init file by using the function **zwei:preload-zmail**.

zwei:preload-zmail &rest *files* *Function*

Starts up Zmail, loading in *files*.

```
(zwei:preload-zmail "wombat:>kjones>mail.text")
```

This gets the mail loading operation underway while you are doing something else.

These are the keyword options to **zwei:preload-zmail**:

:find-file Find the file and load it in for processing.

:examine-file Finds the file and reads it into Zmail but in read only mode.

As an example, the following form can be included in your LISPM-INIT to preload several mail files into Zmail with some of them being read only:

```
(zwei:preload-zmail '(:find-file "y:>palter>mailboxes>palter.xmail")
                    '(:find-file "y:>palter>mailboxes>reminders.xmail")
                    '(:examine-file "y:>palter>mailboxes>junk.xmail")
                    '(:examine-file "y:>palter>mailboxes>digest.xmail"))
```

:hang-when-deexposed

Controls the use of the Zmail background process. Zmail reads and parses the files in the background. If **(:hang-when-deexposed t)** is included at the end of the **zwei:preload-zmail** form, the Zmail background stops after reading the mail files in question without parsing the contained messages. The background parsing will commence as soon as Zmail is selected. The default for **:hang-when-deexposed** is **nil**, so use of **zwei:preload-zmail** without specifying **:hang-when-deexposed** causes mail parsing to begin in the background as soon as the loading is finished.

As an example of the use of **:edit-all-mail-files**, the form

```
(zwei:preload-zmail '(:examine-mail-file #p"LARRY-BIRD:>Palter>mailboxes>digests.kbin")
                    :edit-all-mail-files)
```

will preload Palter's digest `kbin` file with saving disabled and then preload all the other mail files listed in his profile.

Zmail's *Edit File* command, which is used to ask Zmacs to edit a file usually referenced by the current message, is bound to `c-X c-sh-F` at top level.

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.

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 which 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 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".

2. Set the value of the special variable **cp:*dispatch-mode***.

cp:*dispatch-mode* *Variable*

The current Command Processor dispatch mode in Lisp Listeners and **break** loops. Possible values are **:form-only**, **:form-preferred**, **:command-only**, and **:command-preferred**. The meanings of these values is disucssed in the section "Setting the Command Processor Mode". The default is **:command-preferred**.

The default dispatch mode for **cp:read-command-or-form** is the value of **cp::*default-dispatch-mode***.

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".

2. Set the value of the special variable **cp:*prompt***.

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".

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 **cp:read-command** and **cp:read-command-or-form** is the value of **cp::*default-prompt***.

Setting Command Processor Special Characters

You can change the command and form dispatch characters by setting the special variables **cp::*command-dispatchers*** and **cp::*form-dispatchers***.

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 **(#\:)**.

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 **(#\,)**.

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.

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 **cp:read-command** and **cp:read-command-or-form** is the value of **cp::*default-blank-line-mode***.

si:*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 **(flavor:method :start-typeout si:interactive-stream)**.

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".

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". For a description of the **setf** macro: See the macro **setf**.

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.

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.

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.

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.

Init File Form: Setting Refind File to Not Query for Newer Version of File

```
(SETF ZWEI:*REVERT-UNEDITED-BUFFERS-FOR-NEW-VERSION* :ALWAYS)
```

Controls the prompting behavior of Refind File, Refind All Files, and Revert Buffer if a newer version of the buffer file exists on disk. Its default is **:query**, which means ask you if you would prefer the newer version. It may be set to **:always**, meaning pick up the newer version without bothering to ask, or **:never**, meaning do not pick up the newer version.

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).

Fixing White Space For Kill/Yank Commands

```
(SETF ZWEI:*KILL-INTERVAL-SMARTS* T)
```

This tells the killing and yanking commands to optimize white space surrounding the killed or yanked text.

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.

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.

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.

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.

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).

m-. on m-Left

```
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.

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.

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 uppercase.

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.

Customizing the Input Editor

Genera maintains histories of your interactions for you. (See the section "Types of Histories".) You can make use of these histories with the various yanking commands, c-Y, c-m-Y, c-m-sh-Y, m-Y and m-sh-Y. You can control the behavior of these yanking commands with two variables:

zwei:*history-menu-length*
zwei:*history-yank-wraparound*

You can set these with Set Variable (m-X) in Zmacs for the duration of your current session, or you can use **login-forms** and **symbol-value-globally** to set them in your init file.

zwei:*history-menu-length* *Variable*

The maximum number of history elements displayed. Default is 20.

History Menu Length is the name to use with Set Variable (m-X).

zwei:*history-yank-wraparound* *Variable*

Determines what happens when using m-Y reaches the end of a history or m- - m-Y reaches the beginning of a history.

There are two possible actions:

1. If **zwei:*history-yank-wraparound*** is **t**, m-Y wraps around to the other end of the history. For example, after m-Y yanks the oldest element in the history,

it returns to the top of the history and yanks the newest element. This is the default.

2. If **zwei:*history-yank-wraparound*** is **nil**, m-Y stops and the screen flashes or beeps.

History Yank Wraparound is the name to use with Set Variable (m-X).

For more information on the making use of the Input Editor for your own application programs, see the section "The Input Editor Program Interface".

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 press END, this reply is sent 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.

: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 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 **t**, a new message is appended to its conversation.

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 **zwei:*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 **zwei:*converse-end-exits*** to **t** reverses this, so that `c-END` sends the message and remains in Converse and `END` sends and exits.

Customizing Hardcopy Facilities

You can specify the printer you want to use for hardcopying files and screen images in your init file.

There are two variables that determine which printer is used for a hardcopy request:

hardcopy:*default-text-printer* *Variable*

A variable whose value is the printer to be used for printing text files, that is, a printer object. Its initial value is determined from the printer slot in the namespace object for your machine, or if your machine does not specify a printer, from the namespace object for your site.

hardcopy:*default-bitmap-printer* *Variable*

A variable whose value is the printer to be used for printing screen hardcopy, that is, a printer object. Its initial value is determined from the bitmap printer slot in the namespace object for your machine, or if your machine does not specify a bitmap printer, from the namespace object for your site.

These variables can be set with the following two functions:

hardcopy:set-default-text-printer *name* *Function*

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:

```
(login-forms
  (hardcopy:set-default-text-printer "caspian-sea"))
```

`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.)

hardcopy:set-default-bitmap-printer *name* *Function*

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:

```
(login-forms
  (hardcopy:set-default-bitmap-printer "caspian-sea"))
```

`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 your preferred character styles for each printer in your init file by setting **hardcopy:*hardcopy-default-character-styles***.

hardcopy:*hardcopy-default-character-styles* *Variable*

A variable whose value is an association list where each element specifies a device and a set of keyword/value pairs designating the character style. The keywords are **:default-body-character-style** and **:default-heading-character-style**.

For example:

```
(login-forms
  (setq hardcopy:*hardcopy-default-character-styles*
        '(("Itasca" :default-body-character-style
                    (:fix :roman :small))
          ("Caspian Sea" :default-body-character-style
                         (:fix :roman :normal)))))
```

in your init file specifies fixed-width small-sized roman as the default character style for the printer Itasca and fixed-width normal-sized roman as the default character style for the printer Caspian Sea. The value obtained from **hardcopy:*hardcopy-default-character-styles*** is merged with the default style for the printer, so if the printer is using a fixed-width normal-sized roman and you want it larger, you only need to specify **(nil nil :larger)**. See the section "Character Styles".

Customizing FSEdit

There are two variables that you can set to change mouse clicks in the File System Editor and to disable confirmation on deletion:

tv:*use-new-fsedit-command-set* *Variable*

Controls what commands are assigned to the Left and Middle mouse buttons in the File System Editor (FSEdit). **t**, the default, makes clicking Left toggle Open/Close the object and Middle Delete/Undelete the object. Setting **tv:*use-new-fsedit-command-set*** to **nil** restores the pre-Genera 8.0 behavior, where clicking Left opened an object and clicking Middle closed it.

tv:*confirm-fsedit-quick-soft-file-deletion* *Variable*

Controls whether or not deletion in FSEdit requires confirmation on a system that supports *soft* deletion, that is, where undeletion is possible. The default, **t**, is to require confirmation. Setting it to **nil** disables the confirmation. On a file system that does not have soft deletion, FSEdit always asks for confirmation.

Censoring Fields for lispm-finger and name Services

You might prefer to keep certain fields of namespace information private, and prevent those fields from being returned by the **lispm-finger** and **name** protocol servers.

You can censor the information returned by those servers by pushing recognized keywords onto one or both of the following lists: **neti:*finger-fields-to-suppress*** and **neti:*finger-fields-to-suppress-for-untrusted-hosts***.

The recognized keywords include:

:software-info
:hardware-info
:whois
:project
:supervisor
:work-address
:work-phone
:home-address
:home-phone

neti:*finger-fields-to-suppress* *Variable*

This variable is a list of keywords that should be censored for the **lispm-finger** and **name** servers. Use **push** to add others to the list. The default value is **nil**.

For a list of recognized keywords, see the section "Censoring Fields for lispm-finger and name Services".

neti:*finger-fields-to-suppress-for-untrusted-hosts* *Variable*

This variable is a list of keywords that should be censored for the **lispm-finger** and **name** servers, for untrusted hosts only. Use **push** to add others to the list. The default value is **nil**.

For a list of recognized keywords, see the section "Censoring Fields for lispm-finger and name Services".

Logging In Without Processing Your Init File

Sometimes you 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 or try a new system feature in the standard 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.

Customizing Zmail

The Profile command allows you to customize Zmail by setting various display and command options to your personal taste.

You can also customize the Zmail user interface by using the (m-X) Set Key command to temporarily bind a Zmail command to a keystroke. For example, you can use Set Key to temporarily bind the Append Conversation by References command to s-S.

You can set an option temporarily or permanently, the latter by saving the option in your *Zmail Profile*.

Classes of options you can set in your *Zmail Profile* 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
- Clicking Middle 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 109) shows the text of your profile and the current settings of various options.

Setting and Saving Zmail Options

Option settings are stored in seven distinct places:

1. *The Zmail environment:* the way the options are actually set at the moment.

2. *The defaults:* the way the options are actually set before you alter them.

3. *The editor buffer:* the in-memory buffer of your profile.

4. *The source version of your profile:* on disk.

5. *The compiled version of your profile:* also on disk.

6. *Mail buffers:* options associated and stored with the individual mail buffers.

7. *Mail files:* options associated with a mail buffer saved as a file.

Enter profile mode by clicking on [Profile] in the Zmail menu. The top portion of the window looks like Figure 109. The User Options pane works like an Accept Variable Values menu. You click on the various values to change options. The boxed items above and below the User Options pane are menu items that bring up additional menus for general operations on your mail files or on your profile.

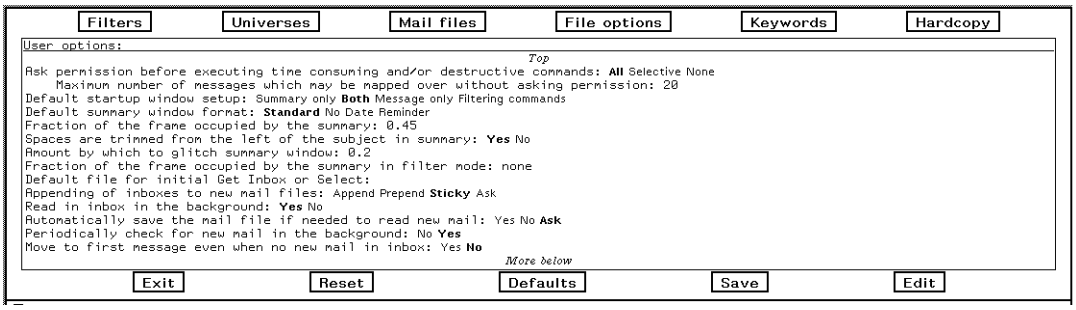


Figure 160. Profile Mode Menu and Interaction Pane

The lower half of the window is an editor buffer into which Lisp forms are inserted automatically when you select options in the User Options pane. This is what is saved as your zmail-init.lisp file. You do not have to edit this file yourself; Zmail takes care of that for you.

The simplest way to use profile mode is:

1. Make the changes you want using the menu items or user options window. For a list of the various options and what they mean: See the section "Zmail Profile Options".

2. Click on [Exit] to leave profile mode. Check to see that you like your changes.

3. To save your changes, reenter profile mode and click Left on [Save]. Answer *yes* to any questions about inserting changes or recompiling your file. At this point Lisp code corresponding to your option settings is stored in your profile. Options changed using [File options] or [Keywords] are stored in the individual mail buffers and are saved when you save the particular mail file.

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 undo all the settings you have made by clicking on [Defaults], which returns all the variables to their system defaults. You can reset all the variables to the values in your init file by clicking on [Reset], which loads your init file again.

For inserting a bug-report banner into the text of bug messages, see the variable **dbg:*character-style-for-bug-mail-prologue***.

Getting Help

The Genera 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 Genera.

This chapter is a collection of the support tools and facilities available for finding the kind of information 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 Genera effectively.

HELP Key

The key labelled HELP looks up context-dependent documentation. Pressing HELP almost always displays something useful. For example, if you have typed se to the Command Processor, and then are not sure what command you actually want, pressing HELP displays:

```
These are the possible command names starting with "se":
  Select Activity
  Send ... (2)
  Set ... (20)

Command: se
```

You can use HELP in combination with other keys to get information about those keys or about Genera.

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.
m-HELP	Shows a helpful hint (*HackSaw*) for getting things done faster and more easily in Genera.

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

c-? and c-/

When you are not sure of the exact name of a command, there are two keystrokes that search the set of available commands and locate possibilities for you. c-? searches for commands whose names begin with the string you have typed so far. c-/ searches for commands whose names contain the string anywhere.

For example, in a Lisp Listener, typing start and pressing c-? yields:

```
These are the possible command names starting with "start":
  Start GC
  Start Printer
  Start Process
  Start X Screen
```

Typing start and pressing c-/ yields:

```
These are the command names containing "start":
  Restart Printer Request  Start GC       Start Process
  Restart Process          Start Printer  Start X Screen
```

Interaction with Completion and Typeout Windows

The Genera 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.

HELP Key in Any Zmacs Editing Window

The HELP key enables you to locate help material that is relevant to the current context. Individual programs are responsible for providing the routines that support the HELP key. The most complex general help facility is that provided by Zmacs editing windows. The HELP key provides access to a number of distinct kinds of help, depending on the key you press after the HELP key.

HELP ? or HELP HELP
: Displays a brief summary of the Zmacs help options (similar to the rest of this chart).

HELP A
: Looks up all Zmacs commands whose names contain a substring you type. Zmacs displays the one-line documentation for the command and tells you which key, if any, invokes it in the current context. The "A" stands for "apropos". When people say, "Use Apropos," they are referring to this command.

HELP C
: Looks up which command is bound to a particular key. You type the key; Zmacs displays the name of the command and its summary paragraph. HELP C uses Self Document.

HELP D
: Looks up the summary paragraph for a Zmacs command. You enter the command name. Completion is available. HELP D uses Describe Command.

HELP L
: Displays a representation of the last 60 keys that you pressed. It is useful to find out what you did that caused unexpected behavior. HELP L uses What Lossage.

HELP U
: Undoes the last operation. You can revert to a prior state by using HELP U or c-sh-U. Zmacs queries you whether to go ahead with undoing; the only information you have about what is being undone is the name of the class of operation, for example, "fill" or "sort". HELP U uses Undo.

HELP V
: Looks up all Zmacs user variables whose print names contain a substring you type. Zmacs displays the variable names and their current values. HELP V uses Variable Apropos.

HELP W
: Finds the key assignment for a particular command. You type the command name; Zmacs displays the current key assignment. Completion is available. HELP W uses Where Is.

HELP SPACE
: Repeats the last HELP command you used.

In this chapter, all Zmacs commands appear by name rather than by key binding. Command tables indicate whether the command has a standard key binding or whether it must be used as an extended command. For example, Edit Zmacs Command is an extended command and requires that you invoke it with m-X. Forward Word is bound to m-F; you invoke it by holding down the META key and pressing F.

Command	*Type of command*
Edit Zmacs Command (m-X)	An extended command
Forward Word (m-F)	A command with a standard key binding
Find File (c-X c-F)	A command with a standard key binding

Functions and their arguments appear as in the following example:

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

Words in italics are the arguments to the function. The words reflect the meaning of the argument. Bold words are optional arguments; you can leave them out. The reference description for the function explains the meanings of the arguments and the default values for optional arguments.

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.

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. This is for commands *beginning* with the characters you have typed.
c-/	Runs Apropos for each of the partially typed words in the name.
Mouse-R	Pops up a menu of the possible completions.
COMPLETE	Displays the full command name, if possible.
RETURN, END	Confirm the command when possible, whether or not you have seen its full name.

You can request completion by pressing 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 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 and 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. Pressing HELP or c-? shows the list of completions in the typeout area; the names are mouse sensitive. Clicking Right 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 "show" or "set".

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 **zwei:edit-definition-internal** (assuming that Zmacs had previously parsed the source file containing it into a buffer).

Completion in Other Contexts

Completion is available in many 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.

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.

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.

Getting help:

- For more information about help facilities in editing, see the section "Getting Out of Trouble".
- For more information about help facilities in the mail program, see the section "Using Zmail".
- For more information about the FEP and a listing of available commands, see the section "The Front-End Processor".

Reference Description of Help Functions

This section contains a summary paragraph of documentation for each of the information-finding commands and functions. See the section "Summary of Help Functions in Zmacs and Lisp".

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, Compile Buffer (m-X) 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 that you can always use HELP W to find the key that invokes a particular command.)

Some Zmacs commands come in pairs, such as 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 (m-X), HELP A
: 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 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 the notation "Bound". **zl:apropos** returns the symbols that it found as a list.

```
(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-F See Show Flavor Initializations.

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 cur-

rent 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-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 **zl: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.

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 saved (the default).
2 For each buffer, since it was last read.
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 saved (the default).
2 Since the buffer was last read.
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.

(si:flavor-allowed-init-keywords *flavor-name***)**

Returns a list containing the init keywords and initable 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.

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 saved (the default).
2 For each buffer, since it was last read.
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.

(compiler:load-compiler-warnings *file* ***flush-flag*****)** (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, **nil** 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.

Show Flavor Initializations (c-sh-F)
: Displays the initializations for the current flavor.

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".

(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". See the section "Trace".

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

Tracing very common functions (like **zl: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* ***package*****)**
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* ***package inferiors superiors*****)**
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)
```

Additional Zmacs Commands:

Add Patch Changed Definitions Of Tag Table
Adds any definitions that have changed in any of the buffers.

Compile Changed Definitions Of Tag Table
Compiles any definitions that have changed in any of the buffers.

Edit Callers In System
: Prepares for editing all functions in the specified system that call the specified function.

Edit System Files
: Reads all the files of a system into buffers.

Find Files In Tag Table
: Reads all files in the current tag table into the editor.

List Callers In System
: Lists all functions in the specified system that call the specified function.

Multiple Edit Callers In System
: Prepares for editing all functions in the specified system that call any of the specified functions.

Multiple Edit Callers Intersection In System
: Prepares for editing all functions in the specified system that call all of the specified functions.

Multiple List Callers In System
: Lists all functions in the specified system that call any of the specified functions.

Multiple List Callers Intersection In System
: Lists all functions in the specified system that call all of the specified functions.

Next File
: Moves to the next file in the tag table, skipping the remainder of the current file.

Select All Buffers As Tag Table
: Selects all buffers currently read in.

Select Some Files As Tag Table
: Selects some files as a tag table. Read successive pathnames from the minibuffer.

Select System Change Buffer
: Selects the system change buffer for the current system change.

Select System Version As Tag Table
: Selects all the files in a specified major version of a system as a tag table.

Select Tag Table
: Makes a tag table current for commands like Tags Search.

Show All Section Changes Of System
: Displays the changes to all buffer sections for all files in a system.

Tags Edit Compare Differences
: Sets up `c-.` to visit all of the compare differences in the current tag table.

Tags Find Pattern Moves to next occurrence of the given pattern within files.

Tags Multiple Query Replace
Replaces occurrences of any number of strings with other strings within the tag table.

Tags Multiple Query Replace From Buffer
Replaces occurrences of any number of strings with other strings within the tag table.

Tags Query Replace
Replaces occurrences of one string with another within the tag table files.

Tags Spell Reads through the buffers of the tag table and collect misspellings into a buffer.

Take Merge Choice Prompts for a tag, and inserts the text for that tag into the merge results.

Visit Tag Table Reads in the specified tag table file.

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. The input editor is a feature of all interactive streams.

For a complete list of the commands available in the input editor, press c-HELP or see the section "Input Editor Commands".

Histories and Yanking

A *history* remembers commands and pieces of text, placing them in a history list. Additions to the history are placed at the top of the list, so that history elements are stored in reverse chronological order — the newer elements at the top of the history, the older elements toward the bottom.

Yanking commands pull in the elements of a history. *Top-level commands* start a yanking sequence. Other commands perform all subsequent yanks in the same sequence. A yanking sequence ends when you type new text, execute a form or command, or start another yanking sequence.

The system has different histories for different contexts. One of these is always the *current history*.

Types of Histories

Genera uses the following histories:

Input	History containing text typed at the input editor; a separate history exists for each window.
Kill	History of text deleted or saved in any window; a global history.
Replace	History of arguments to Query Replace (m-X) and related commands.
Buffer	History of editor buffers visited in this window.
Pathname	History of file names that have been typed.
Command	History of editor commands that use the minibuffer, and their arguments. Commands that do not use the minibuffer, such as m-RUBOUT, are not recorded in the history.
Definition	History of names of definitions that have been typed.

Except for the input histories, which are per-window, only a single instance of each of these histories exists, shared among all editors, including Zmacs, Zmail, and Dired.

Input Editor

In the input editor c-m-Y yanks from the history of previous inputs.

Because the input editor's kill history is the same as the Zwei kill history, c-SPACE, c-W, m-W, c-<, c->, c-Y, and related commands can be used in the input editor to move text back and forth between Zmacs, Zmail and the Input Editor. (Press c-HELP for a summary of commands.) Unlike Zwei, however, the input editor does not underline a marked region.

You can use most Zwei editing commands on yanked forms. Reactivating a yanked form is simple: just press END anywhere within or at the end of the form. ESCAPE displays the history of previous inputs. A numeric argument controls the length of the input history to be displayed. An argument of 0 displays the entire history.

c-ESCAPE displays the default kill history. A numeric argument controls the length of the kill history to be displayed. An argument of 0 displays the entire history.

The Displayed Default

When a command that reads an argument in the minibuffer displays a default, it puts the default onto the history temporarily. After reading and defaulting your input, it puts the argument onto the history instead. Thus c-m-Y always yanks the displayed default and c-m-2 c-m-Y yanks the last thing typed in that context. If no default is displayed, c-m-Y yanks the last thing typed in that context.

The displayed default is usually not the same as the most recent item in the history; often it is computed according to some heuristic based on past history and

the exact command being given. It is pushed onto the top of the history in order to allow you to easily yank and edit it. This is useful when the heuristic comes close but does not provide exactly what you want.

Using Numeric Arguments for Yanking

A numeric argument of 0 to any yank command displays a list of the history and the numeric argument required to get each element of the history.

Example: The input history invoked in a Dynamic Lisp Listener by c-m-0 c-m-Y:

```
Lisp Listener 1 Input history:
  1: (+ 210 32)
  2: (* 17 6)
  3: Load Patches
  4: Show System Modifications
  5: Show Herald
  6: Login KJones
```

The history is displayed in reverse chronological order — the newest element first, for example, (+ 210 32); the oldest last, for example, Login KJones.

By default, a positive argument to c-Y and c-m-Y specifies how far from the newest element into *past* history is the element to be yanked. The numbers in the history display can be used as numeric arguments. (Optionally, you can set the variable **zwei:*history-rotate-if-numeric-arg*** so that arguments to the yanking commands are measured relative to the origin. See the section "Customizing the Input Editor".)

Example: c-m-1 c-m-Y yanks element #1, (+ 210 32), from the history.

Example: c-m-2 c-m-Y yanks element #2, (* 17 6), from the history.

A positive or negative argument to m-Y is measured relative to the last element yanked, not the newest element.

Example: Pressing c-m-2 c-m-Y yanks (* 17 6); then pressing m-4 m-Y yanks Login KJones, not element #4. Displaying the history at this point looks like this:

```
Lisp Listener 1 Input history:
   1: (+ 210 32)
   2: (* 17 6)
   3: Load Patches
   4: Show System Modifications
   5: Show Herald
-> 6: Login KJones
```

Element #6, marked by a pointer, is the origin. (Note: The origin is not the most recent element because m-Y has changed the origin.)

A top-level command given without an argument retrieves the element at the origin, which is the last element yanked in the previous yanking sequence, not necessarily the newest element of the history.

Example: c-m-Y yanks Login KJones) from the history.

A numeric argument of c-U not followed by any digits is the same as no numeric argument with one exception: Point is placed before the text yanked and mark is placed after — the reverse of the ordinary placement.

To find out how to customize the input editor, see the section "Customizing the Input Editor".

System Conventions and Helpful Hints

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 **zl:read** function), and not the person reading this documentation.

By default, Symbolics Common Lisp displays numbers in base 10. If you wish to change it: See the section "What the Reader Recognizes".

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 the printed representation of a lisp object; "Dotted pair", which means a cons; and "Atom", which means, roughly, symbols and numbers and sometimes other things, but not conses.

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".

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 Genera system. Every site has a copy of the basic 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. See the section "Logical Pathnames".

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**.

What is a World Load?

A world load can generally 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 **(zl:disk-save)**.

World loads are stored in world-load files. These are FEP files with the file type `.load`. There are three kinds of world-load files:

- Complete worlds that can be loaded and run as is. If you load a world, load MACSYMA, and then save the whole world, you have a complete world as the result.

- IDS (Incremental Disk Save) worlds that can have parents and offspring. If you load a world without MACSYMA in it, load MACSYMA, and do an incremental save, you have an IDS world as the result. When you load an IDS world, it calls for its parents and loads them as well.

- Netboot cores that are used to effect netbooting. Netboot cores are very small world-load files that can seek out worlds on remote netboot server machines and boot those worlds from there.

What is a Netboot Core?

A netboot core is a world load consisting of the restricted set of capacties necessary to netboot. A netboot core can find a world on a netboot server and boot it from there. A netboot core file is usually about 100 blocks. Any Genera 7.2 or later world-load file can be used *as* a netboot core.

For more information: See the section "Netbooting".

How to Delete and Expunge a File Left Open by an Aborted Restore Distribution

If your machine halts at an inopportune time, for instance when a file is open for writing during a Disk Restore, you might leave LMFS's data structures in an inconsistent state; `c-m-ABORT` or, in some circumstances, `c-ABORT` can do this, especially on Ivory machines. If the internal structure is inconsistent LMFS is not able to Delete and Expunge. This leaves the file marked with a `W`. Deleting and expunging seems to have no effect. The file has length 0, and you cannot open it because it is already open. [Close All Files] doesn't affect the file's status. Try a warm boot (of the server machine, of course). After a boot, LMFS's datastructures are rebuilt. The file is still marked with a `W` because LMFS knows that it was in the middle of being written, but you can delete and expunge it normally.

Why do you name machines and printers?

Naming inanimate objects such as hosts, printers, sites, and networks may seem foolish if you have only one 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 machines after the wives of Henry VIII.

Why Does My Machine Sometimes Crash During a GC When Show GC Status Says There is Enough Room?

The estimate of how much space the Garbage Collector is going to need is just that, an estimate. If you have not done a garbage collection before in this boot session, and if you boot a world from your local FEP, it is difficult to estimate how many dynamic objects there are in your world due to the effects of swap migration on how much copy space is needed. For the details of swap migration, see the section "Swap Migration and Garbage Collection".

Questions about the FEP and LMFS

Why can't I write out files when I have free disk space?

The disk attached to a Symbolics machine 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 Symbolics 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 *Symbolics Machine Name:>directory>filename.type.version*. The entire Genera 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 LMFS Partition".

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 Editing Operations window, by using `SELECT F`, by clicking on [File System] in the System menu, or by using the Select Activity File System Operations command. Then click on [Local LMFS Operations]. The second-level menu pops up. Click on [LMFS Maintenance Operations]. 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 Create FEP File command creates a FEP file for purposes other than a LMFS partition. See the section "Create FEP File Command". The following Lisp function also creates such a FEP file:

```
(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 Systems".

HackSaws

HackSaws are interesting, often little known, facts about Genera. In some cases, they are things that have passed into folklore and, while documented, are often not emphasized or obvious to a newcomer to the machine. The name comes from *hack*, a neat trick, and *saw*, a familiar saying. They are useful commands, functions, tools, and workstyle advice that expert users of Symbolics computers usually know and use automatically but do not think of telling a new user about. Often there is no very good way to categorize or index these HackSaws, thus making it difficult to present them coherently to a user.

The HackSaw facility provides a file of these facts, accessed by the Show HackSaw command, which is bound to m-HELP for ease of typing. m-HELP opens the file and selects a HackSaw at random, displaying it on your screen.

The commands for displaying HackSaws and for adding your own site-specific or personalized file of hints and interesting facts are:

Show HackSaw Command
: Shows an interesting fact about Genera.

Add HackSaw Command
: Lets you add your own interesting facts to the HackSaw files.

Find HackSaw Command
: Finds a HackSaw containing some particular term.

Load HackSaw File Command
: Loads all HackSaw files. This is designed to be used with **cp:execute-command** in an init file.

The HackSaws are stored in the file SYS:DATA;HACKSAW.TEXT. It is a plain text file.

The individual HackSaws are separated by a down-arrow, sy-H, on a line by itself.

Although you can edit this file to add HackSaws, you should not do so since when new software releases are distributed, a new version of hacksaw.text is included and you would then have to merge your extensions with the new file. The file SYS:DATA;HACKSAW-EXTENSIONS.TEXT is intended for your use to add your own favorite HackSaws or site-specific facts and hints. You can add items to hacksaw-extensions.text by editing the file directly, being careful to separate individual items with sy-H on a line by itself, or you can use the Add HackSaw command. You can also create your own personal file of helpful hints to be accessed by m-HELP.

To include your own personal file in the HackSaw facility, add it to the list in **approach:*hacksaw-extension-files*** by putting something like this in your init file:

```
(push "Wombat:>KJones>my-hacksaws.text"
      approach:*hacksaw-extension-files*)
```

(See the section "How to Create an Init File".) The initial contents of **approach:*hacksaw-extension-files*** is the file SYS:DATA;HACKSAW-EXTENSIONS.TEXT. When you use Show HackSaw for the first time (or when you use Load HackSaw File), this list is checked and all the files on it are loaded.

If you place the form (cp:execute-command "Show HackSaw") in your init file, you can see a HackSaw automatically everytime you log in.

You can change the key that Show HackSaw is bound to by changing meta-help in the following form to the key of your choice:

```
(si:add-ie-command 'key-bindable-show-hacksaw #\meta-help)
```

The nature of HackSaws means that they are very useful when you need them, but are not always easy to remember if you have not yet had the need for them. If you have seen one go by and then later find a need for that bit of information, you can locate it using Find HackSaw. For example, if you have seen

```
In Zmacs or Zmail, you can click m-Left on a definition name to edit it.
```

You might remember that it was something useful about editing definitions, so you type Find HackSaw definition and Find HackSaw displays all the HackSaws that contain the string definition. Of course, you can also use Find HackSaw to search the HackSaw files for HackSaws on a particular topic.

Recovering From Errors and Stuck States

Sometimes it is hard to know whether or not your machine is in trouble, because some operations, particularly those involving other machines on a network, 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. UX-family machine consoles do not always have run bars, so if you have a UX-family machine you must rely on the process state changing. 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 machine is halted. As long as the FEP is working, it prints a message on the screen when the machine has halted and then gives its FEP Command:> prompt. When the machine resumes its previous state, it updates the clock with the correct time.

Recovery Procedures

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

Wait Forever	Select a different window, then reselect the one you were in.
Output Hold	Press FUNCTION ESCAPE; 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)	Some windows take several seconds to initialize, but if (no window) persists with no run bar activity, use the mouse or SELECT key to select the window you want.

If you are still stuck, 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.

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

The Cold Load Stream

Occasionally an error in the window system and an attempt to abort a process at the wrong instant causes you to go into the *cold load stream*. The cold load stream is the bare Lisp system that underlies Genera and the window system. When you are in the cold load stream, you see a message including the phrase

```
--> ... using the cold load stream <--
```

You still have the full capability of Lisp, you just do not have access to the window oriented display features, including the mouse.

Exactly what you should do to get out of the cold load stream depends on what error caused you to end up there. However, there is usually something you can do. If you are in the Debugger in the cold load stream, you can press c-M to generate a backtrace to be sent as a bug report. After you have successfully exited from the cold load stream and the sheduler has restarted, the bug mail window comes up with your backtrace and you can send the report. To exit from the cold load stream:

- Be sure to read the error message carefully. Some errors give explicit information about what went wrong and how you can recover.

- Process-related errors often have a restart option to arrest the process and restart the scheduler.

After you are out of the cold load stream, your window system might be in a strange state. There are several things to try:

- FUNCTION ESCAPE to get out of output hold.
- FUNCTION c-T to clear any pop-up windows.
- FUNCTION c-CLEAR INPUT to clear all window system locks.
- SELECT L to get back to a Lisp Listener.
- If the mouse process was the problem and you arrested it, use Initialize Mouse to restart it.

You can enter the cold load stream explicitly with FUNCTION SUSPEND. RESUME returns you to the window system.

If you have a UX-family machine and you have been in the cold load stream for more than 60 seconds, when you try to return, you might find that your console window has disappeared. That is due to TCP dropping the connection between the Ux-family machine board and the X-windows application. You can recover by pressing c-C to get out of the genera program. Then run genera again.

Warm Booting

Warm booting is a recovery procedure that may enable you to restart Lisp in order to save buffers and mail. It destroys the state of the process running at the time of the boot, destroys the state of the window system, and resets all network connections. If you warm boot after a crash, do not expect to continue running for very long after a warm boot, unless the cause of the crash can be rectified.

Once you've warm booted, save your work, try to determine the cause of the crash, and cold boot the machine. For information about investigating problems that cause your machine to crash, see the section "Debugging in the FEP".

Warm boot the machine by using one of the following procedures:

1. If the machine crashed, issue the Show Status FEP command at the FEP prompt, check the information it provides, and then issue the Start FEP command. For information about checking the information that the Show Status FEP command provides, see the section "Debugging in the FEP".

2. If the machine did not crash, but is unresponsive, issue the Command Processor (CP) Halt Machine command, or press h-c-FUNCTION if you cannot get a Lisp Listener or if the Lisp Listener is not responding to keyboard input. (Note: on UX-family machines h-c-FUNCTION only works from the cold load

stream, so you might have to first enter the cold load stream with FUNCTION SUSPEND or by opening the cold load icon in order to enter the FEP.) Then, issue the Start FEP command at the FEP prompt. For information about halting Lisp, see the section "Halting". Alternatively, on MacIvory machines, you can choose Restart Lisp from the Ivory menu item to warm boot Lisp.

Note: On MacIvory machines, choosing Shutdown from the Ivory menu, subsequently choosing Quit from the File menu, and then choosing Restart Lisp Ivory menu item also causes a warm boot.

In this case, though, because you have properly shut down Lisp (instead of having crashed), you can expect to operate normally; your machine's state will be as it was before quitting Lisp (rather than initialized, as it would be after a cold boot).

Halting

Halting the machine leaves all Lisp states intact. To halt the machine in order to connect to the FEP, type Halt Machine to a Lisp Listener or press h-c-FUNCTION. You are now connected to the FEP. To return to Lisp, type continue at the FEP prompt (FEP Command>) and press RETURN.

The machine can halt itself under exceptional conditions. In this case, try typing continue. If continue does not work, use start. See the section "Warm Booting".

Resetting the FEP

Resetting the FEP clears FEP memory. You can reset the FEP either from the keyboard or from the processor's front panel. On 3600-family machines, you must reset the FEP if you receive the error message No More Memory.

To reset the FEP from the keyboard:

1. Type the Halt Machine command at a Command Processor command prompt to stop Lisp. If the Command Processor does not respond, press h-c-FUNCTION to stop Lisp.

   On UX-family machines, h-c-FUNCTION only works from the cold load stream, so if the CP does not respond, you have enter the cold load stream by pressing FUNCTION SUSPEND or by opening the cold load icon and then press h-c-FUNCTION.

   On MacIvory machines, you can also click on Transfer to FEP in the Ivory menu.

2. Type the command Reset FEP to the FEP prompt.

3. Press Y to answer the confirmation prompt.

To reset a 3600 machine model from the front panel, press the RESET button and then press the spring-loaded YES switch to answer the "Reset FEP?" query.

To reset a 3610, 3620, 3640, 3645, 3670, or 3675 from the front panel, press the RESET button on the processor front panel.

To reset a 3650 from the front panel, turn the key switch momentarily to RESET.

To reset an XL400 from the front panel, press the RESET button on the processor front panel.

To reset a MacIvory click on `Cold Boot FEP` in the Ivory menu.

The only way to reset the FEP on a UX-family machine is with the Reset FEP command.

Type the Hello command to the FEP prompt, give the Start command, and press `RETURN` to warm boot the machine and to to Lisp.

How to Get Output to a Printer

Introduction to the Hardcopy Facilities

The Hardcopy system provides a uniform interface for sending output to a printer. It allows the user or the program to specify formatting information in a device independent way for output on a supported printer.

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.

Printing and Hardcopy Commands

Commands for Producing Hardcopy

You can produce hardcopy from the Command Processor, by 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.

You can hardcopy topics from Document Examiner. See the section "Document Examiner".

Hardcopying from the Command Processor

Hardcopy File **Command**

Hardcopy File *pathname printer keywords*

Sends a file to a hardcopy device.

pathname The pathname of the file you are printing. You can specify more than one file by separating the pathnames with commas.

printer The printer on which you are printing the file. The default is the default hardcopy text printer (the value of **hardcopy:*default-text-printer***).

keywords :Body Character Style, :Copies, :Delete, :Ending Page, :File Types, :Heading Character Style, :Notify, :Orientation, :Print Cover Page, :Running Head, :Starting Page, :Title

:Body Character Style
The character style used for printing the text of the file. You can also use this character style for merging character styles in the file. See the section "Understanding Character Styles".

:Copies {*number*} The number of copies you are printing. The default is 1.

:Delete {Yes, No} Enables you to specify whether the file deletes after printing. The default is No, not to delete.

:Ending Page {*number*} The last page you are printing. This command defaults to the last page of the file. Note that the page character or form feed character identifies page breaks. The system treats text files without page breaks as a single page, although the file can use many sheets of paper.

Press format files contain form feeds or PAGE characters. Remember that these are physical pages and do not necessarily correspond to the page numbering appearing in the heading. For example, the first page of a press file is a title page, the second page is numbered *i*, and the third page is numbered 1.

:File Types {ASCII, DMP1, LaserWriter, PostScript, Press, Text, XGP, use-canonical-type} The internal format of the contents of the file, to interpret for printing. The default is use-canonical-type, meaning that the type is determined from the extension to the file name.

:Heading Character Style
The character style used for the running head.

:Notify {Yes, No} Specifies whether to send a notification upon job completion. The default is Yes.

:Orientation {Landscape, Portrait} Specifies the paper orientation for the output. Portrait is left to right across the short dimension of the paper. Landscape is left to right across the long dimension of the paper. The default is Portrait.

:Print Cover Page {Yes, No} Specifies whether to print a cover page. The default is Yes.

:Running Head — {None, Numbered} Specifies the type of running head that prints on the top of each page. The default is Numbered.

:Starting Page — {*number*} The first page you are printing. This command defaults to the first page of the file. The page character or form feed character identifies page breaks. The system treats text files without page breaks as a single page, although the file can use many sheets of paper.

Press format files contain form feeds or PAGE characters. Remember that these are physical pages and do not necessarily correspond to the page numbering appearing in the heading. For example, the first page of a press file is a title page, the second page is numbered *i*, and the third page is numbered 1.

:Title — {*string*} Specifies the title appearing on the cover page identifying the output. The default is "File: *pathname-you-specified* ".

Hardcopying From the System Menu

You can produce hardcopy by clicking on [Hardcopy] in the System menu. The Accept Variable Values menu appears, enabling you to specify the pathname of the file you are hardcopying, and to select the printer, character style, and the other parameters offered by the Hardcopy command (see Figure 135).

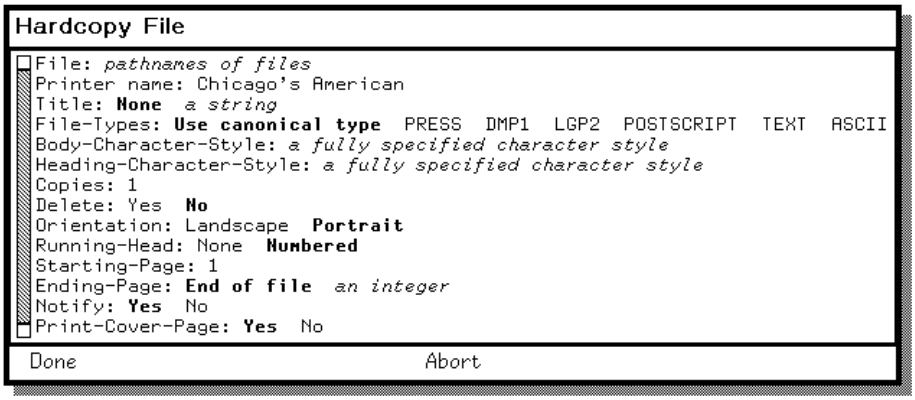


Figure 161. The Hardcopy Menu

Hardcopying From Zmacs

You can use the following commands when hardcopying from Zmacs:

- Hardcopy Region Command
- Hardcopy Buffer Command
- Hardcopy File Command
- Kill or Save Buffers Command

Hardcopy Region Command

Hardcopy Region (m-X)

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

For full information on Genera hardcopying, see the section "How to Get Output to a Printer".

Hardcopy Buffer Command

Hardcopy Buffer (m-X)

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

For full information on Genera hardcopying, see the section "How to Get Output to a Printer".

Hardcopy File

Hardcopy File (m-X)

Enables you to specify the name of a file for printing on a local hardcopy device.

For full information on Genera hardcopying, see the section "How to Get Output to a Printer".

Kill or Save Buffers Command

Kill Or Save Buffers (m-X)

Displays a menu listing all existing buffers. Modified buffers are initially marked for saving. Choices are: Save, Kill, Unmodify, and Hardcopy. Specify these options next to the buffer names in the menu. This command is bound to c-X c-m-B and appears on the editor menu. Specifying a numerical argument to c-X c-m-B inhibits the initial marking of the menu.

Hardcopying From Zmail

You can use the following commands when producing hardcopy from Zmail:

- Hardcopy Message Command
- Hardcopy All Command
- Show Printer Status Zmail Command
- Hardcopy File Zmail Command
- Format File Zmail Command

Hardcopy Message Command

Hardcopy Message (m-X)
: Hardcopies the current message. Note that you can also click Right on [Other] in the Zmail menu and select Hardcopy Message. Additionally, you can click Right on the message summary line, and then click Right on [Move] and select Hardcopy.

Hardcopy All Command

Hardcopy All (m-X)
: Hardcopies all the messages in the current sequence. Note that you can also click Right on [Map Over] and select [Hardcopy] for copying all messages in the current sequence.

Note that whether you hardcopy a single message or hardcopy all messages in a sequence, you can click Right on [Hardcopy] and specify the number of copies and which printer to use. The `Other` option in the list of printers enables you to specify an arbitrary printer, using either the pretty name or the namespace name. This printer becomes the selected printer, and remains in the menu for subsequent hardcopy commands.

Show Printer Status Zmail Command

Show Printer Status (m-X)
: Prompts for the name of a printer and displays its print queue.

Hardcopy File Zmail Command

Hardcopy File (m-X)

Sends the file associated with the pathname you specify to the default printing device. The default is the first pathname specified in the `File-References:` header field. If there is no `File-References:` field, the default is the current mail file.

Format File Zmail Command

Format File (m-X)

Formats the file associated with the pathname you specify. c-U m-X Format File formats the file and sends it to a printer. The default pathname is the first pathname specified in the File-References: header field. If no File-References: field exists, the default is the current mail file.

Hardcopying from Dired

You can hardcopy files in Dired by marking files. When you exit Dired, the marked files 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 is marked.

You can specify a numeric argument *n*, marking the next *n* files for printing.

Hardcopying the Screen

You can produce a hardcopy of your screen by pressing FUNCTION Q or FUNCTION *n* Q (where *n* is any numeric argument):

Function Q	Captures a screen image for hardcopying or inserting in a file.
Function *n* Q	(where *n* is any numeric argument, for example FUNCTION 0 Q) Displays a menu of options for capturing screen images. Note that the choices you make remain in effect for subsequent uses of FUNCTION Q.

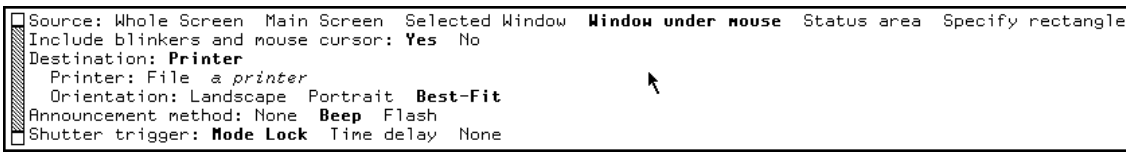


Figure 162. FUNCTION 0 Q

Source: options identify the area of the screen to capture:

Whole Screen	A bitmap image of the entire screen

Note that a full-screen bitmap image includes a border around the actual screen image. If you do not want this extra whitespace around the image, select the Named Image option and edit the bitmap image using the "Refit Bounding Box Bitmap Editor Command".

Main Screen — All but the status area

Selected Window — The window of the selected activity (for example, Symbolics Concordia)

Window Under the Mouse
The window under the mouse cursor

Use this option when you cannot use the mouse to specify a portion of the screen (for instance, to take a snapshot of a pop-up menu).

Status Area — The window at the bottom of the screen (including the mouse information line(s))

Specify Rectangle — Specify the area of the screen to be captured using the mouse

Window History — The history of the specified window

`Shutter Trigger:`

Mode Lock — Captures the image when the `MODE LOCK` key is pressed. This allows you to *set up* the screen a certain way, perhaps with a menu showing, or the mouse cursor in a certain place.

Time Delay — Allows you to specify a time delay before capturing the image. The default is 5 seconds.

None — Captures the image as soon as you click on `[Done]`. This is the default.

Hardcopying from the File System Editor

You can produce hardcopy using the system hardcopy menu from FSEdit by following these steps:

1. Click Right on a file name. A menu of file operations appears.
2. Click on `Hardcopy` in the menu of file operations. The menu in Figure 25 appears.
3. You can modify any of the parameters displayed. Clicking on `Done` prints the file.

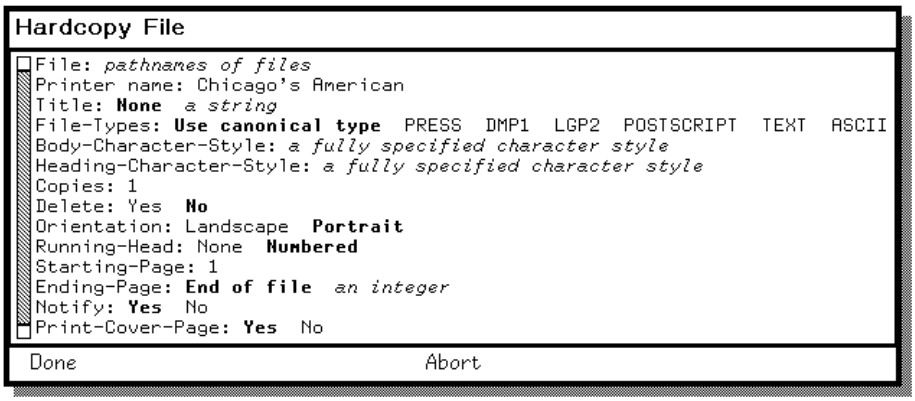


Figure 163. The Hardcopy Menu

Other Hardcopy Commands

Changing the default printer

You can change the default printer using the Set Printer Command.

Set Printer **Command**

Set Printer *printer-name keywords*

Sets the default printer for hardcopy.

printer-name	The name of a supported printer that can be reached by your machine.
keywords	:More Processing, :Output Destination, :Output Type
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, Printer, Stream, Window} Enables you to direct the output of this command. The default is the stream ***standard-output*.**

:Output Type {text bitmap both} Enables you to specify whether a printer prints text (files and mail messages), bitmap (graphics and screen hardcopy), or both text and bitmap.

Additional Methods of Changing the Default Printer

- You can change the default printer in your init file by specifying the printer most convenient for you. See the function **hardcopy:set-default-text-printer**.
- The Hardcopy File command accepts a keyword argument of :printer enabling you to specify a printer. For example:

```
Hardcopy File q:>kjones>report.pr :printer beacon
```

- Additionally, you can specify a different printer by clicking on the printer name.
- The System menu enables you to specify a different printer by clicking on the printer name.

You can view the default printer using the Show Printer Defaults command.

Show Printer Defaults **Command**

Show Printer Defaults *keywords*

Displays the current default printer(s). If you send all your hardcopy output to one printer, the command returns:

```
Default Printer (for both text and bitmap output): printer-name
```

If you use a different printer for text and screen hardcopy, the command returns:

```
Default Text Printer: printer-name1
Default Bitmap Printer: printer-name2
```

Managing the Print Queue

You can use the following commands for managing the print queue:

- Show Printer Status Command
- Delete Printer Request Command
- Restart Printer Request Command
- Halt Printer Command

- Start Printer Command
- Reset Printer Command

Show Printer Status **Command**

Show Printer Status *printer*

Displays the print queue for the specified printer or printers.

printer — The name of a printer or printers (separated by commas) whose print queue you are displaying. Specifying All displays the queues for all printers at your site. The default is your current printer. If your text printer and your bitmap printer are different, the command uses your text printer as the default for Show Printer Status.

The display of requests is mouse sensitive, enabling you to click on select arguments when using either the Delete Printer Request or Restart Printer Request commands.

Note that this command is also available in Zmail as `m-X` Show Printer Status.

Delete Printer Request **Command**

Delete Printer Request *printer-request*

Deletes the specified print request from the print queue.

printer-request — A string specifying the printer and the request you are deleting. You can select the print request with the mouse from the display of the Show Printer Status command. For more information, see the section "Show Printer Status Command". You can also specify the printer name to which you are sending requests and use "`c-?` and `c-/`"for displaying all requests. For more information, see the section "Completion in the Command Processor".

keywords — :Confirm

:Confirm — {Yes, No} Whether to request confirmation when you delete a request currently printing. The default is Yes.

Restart Printer Request **Command**

Restart Printer Request *printer printer-request keywords*

Restarts a print request that you stopped before completion. For requests currently printing, the printer resets and the request prints from the beginning. For requests on hold, the printer requeues the request.

printer-request	A string specifying the request. You can select the print request with the mouse from the display of the Show Printer Status command. You can also specify the printer name to which you sent the request and use "c-? and c-/" for displaying all requests. For more information, see the section "Completion in the Command Processor".
keywords	:Extent, :Starting From
:Extent	{Entire, Copy} The extent to which you are restarting a request. Entire refers to the whole request. Copy refers to a single copy.
:Starting From	{*number*} The number of the copy after which the printer restarts. Note that you cannot use this keyword if you used the :extent keyword specifying Entire. The default is 0. *This option is currently not used.*

Halt Printer **Command**

Halt Printer *printer keywords*

Halts the specified printer. The printer does not print any requests until you start it using the "Start Printer Command".

printer	The name of the printer you are halting.
keywords	:Confirm, :Disposition, :Reason, :Urgency
:Confirm	{Yes, No} Whether to ask for confirmation. The default is Yes.
:Disposition	{Delete, Hold, Restart} Enables you to specify whether the system delete, hold, or restart the print request. The Delete option deletes the request from the queue. Hold retains the request in the queue but does not print it when the printer restarts. Restart automatically prints the interrupted request from the beginning when the printer restarts. The default is Hold.
:Reason	{*string*} Enables you to specify the reason for the shutdown. This option appears in the display of the Show Printer Status command and in the Print Spooler log. The default message is "Printer *printer-name* being reset by *user-id*."

You can use the following keyword to control precisely when the printer halts:

:Urgency — {Asap, After-Current-Request, After-Next-Copy} Enables you to specify when the printer halts. Asap indicates that the printer halt and reset immediately. After-Current-Request indicates that printer halt after the current request finishes printing. After-Next-Copy indicates that the printer halt after the next copy of the request prints. After-Next-Copy is the same as After-Current-Reqest when the request is only for one copy. The default is Asap.

When you halt the printer, the Print Manager process enters a suspended state until you start it again.

Note that you can halt, start, or reset a spooled printer from any machine on the network.

Start Printer **Command**

Start Printer *printer*

Starts the specified printer printing its print queue after you halt it.

printer — The name of the printer you are starting.

Note that you can halt, start, or reset a spooled printer from any machine on the network. For more information, see the section "Halt Printer Command".

Reset Printer **Command**

Reset Printer *printer printer-request keywords*

Resets a spooled printer. You can use this command if your printer stops operating or if you have to stop the printer in the middle of a job. This command reestablishes communications with the printer and, when possible, sends a software "reset" command to the printer.

printer — The name of the printer you are resetting.

printer-request — (Optional) Resetting a printer in the middle of a print request results in the system displaying the request and asking you to confirm the reset command. You have to specify the name of the print request or select the print request with the mouse from the display of the Show Printer Status command in order to reset the printer immediately. For more information, see the section "Show Printer Status Command".

keywords — :Confirm, :Disposition, :Reason

:Confirm — {Yes, No} Whether to request confirmation that the printer is printing a request. The default is Yes.

:Disposition — {Delete, Hold, Restart} Enables you to specify whether the system delete, hold, or restart the print request. The Delete option deletes the request from the queue. Hold retains the request in the queue but does not print it when the printer restarts. Restart automatically prints the interrupted request from the beginning when the printer restarts. The default is hold.

:Reason — {*string*} The reason you are resetting the printer, which is added to the printer log. The default message is "Printer *printer-name* being reset by *user-id*."

Note that you can halt, start, or reset a spooled printer from any machine on the network.

When and How to Use the Garbage Collector

Principles of Garbage Collection

It is fundamental to the nature of Lisp that programs and systems allocate memory dynamically and in large amounts. (The allocation of memory for a basic list element, or *cons*, or for any other purpose, is called *consing* for the purpose of this discussion and in most other writings on Lisp.) Even with the large amount of virtual memory on a Symbolics computer, it is possible for a program to use it all up. At this point the machine halts and must be rebooted. This event can always be delayed, almost indefinitely, if the underlying system can reclaim memory that is unused.

Objects that are no longer in use, with no references from other objects, are termed *garbage*. Garbage is distinguished from *good objects* or *good data* by the fact that it no longer serves any purpose in the current Lisp world. For example, if the car of a cons is changed from object A to object B, and there are no other references to A, then A is garbage. Objects in the Genera environment can be said to have a *lifetime,* which means how long the object remains "good". Three lifespans are distinguishable:

Static — Object will probably never become garbage. Example: standard system functions.

Dynamic — Object will probably become garbage eventually. Example: lines in editor buffers.

Ephemeral — Object will probably become garbage very quickly. Example: intermediate structure generated by the compiler.

You can control the garbage collection status of your own areas with the **make-area** function.

Garbage collection (GC) involves these three steps:

- *Scavenging* virtual memory, that is, periodically sifting through areas of memory, separating good objects from the garbage

- *Transporting* good objects to a safe place

- Reclaiming the memory occupied by garbage

Several strategies for garbage collection exist. Some allow you to continue doing other work and some do a more complete job but require additional machine resources for some period of time.

Garbage collection need not be used at all. However, it cleans up after computations and allows you to run for longer periods of time between cold boots. It should be used either when you are running a program that allocates large amounts of virtual memory (where the total allocated might exceed the amount of free memory in a cold-booted system) or when the total allocations of many programs might, over a relatively long period of time, exceed the capacity. In either case, garbage collection is a strategy aimed primarily at preserving the state of an operating Lisp world as long as possible and avoiding a cold boot.

There are three basic modes of garbage collection, each with some variations possible:

- *Incremental garbage collection* works in parallel with other processes in the system, allowing you to continue working while it is in progress. This mode is based on *incremental copying*, so called because objects are copied one at a time and there is relatively little effect on the user's interaction with the system. *Dynamic-object garbage collection* incrementally collects garbage in all *nonstatic* areas of memory. *Ephemeral-object garbage collection* incrementally collects garbage, concentrating on specific parts of memory that are known to contain short-lived objects. Both kinds of incremental operation ignore *static* areas of memory that change slowly and so are unlikely to contain garbage. For an explanation of static memory, see the section "Theory of Operation of the GC Facilities".

- *Nonincremental, or immediate, garbage collection* takes less free memory and less total processor time to work successfully than does the incremental mode. Nonincremental garbage collection is normally done with the Start GC :Immediately command or with the **gc-immediately** function, although those directives still ignore static areas. These directives allow no other work to be done by the process running it, although other processes are still scheduled. In most cases, though, immediate garbage collection places a heavy enough burden on the machine that other processes are not useful while it is operating. The immediate garbage collection invoked by the function **si:full-gc** deals with static areas.

- *In-Place garbage collection* is similar to immediate (nonincremental) garbage collection, but uses a fundamentally different algorithm for storage reclamation. Because of this, the virtual memory (paging space) required for GC is reduced. However, In-Place Garbage Collection is much slower, completely non-

interruptable, and results in less optimal paging behavior than normal Immediate Garbage Collection. In-Place Garbage Collection is typically used only for dynamic objects, but it may be used to reclaim static objects as well.

Note: Areas of memory can be specified as being static with the function **make-area**.

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 METERING:METERING-CONS-AREA: capacity 196K, 0K allocated, 0K
 used.
Second level of METERING:METERING-CONS-AREA: capacity 98K, 0K allocated, 0K
 used.

First level of DW::*EQL-DISPATCH-AREA*: capacity 98K, 256K allocated, 56K used.
Second level of DW::*EQL-DISPATCH-AREA*: capacity 49K, 0K allocated, 0K used.

First level of WORKING-STORAGE-AREA: capacity 196K, 448K allocated, 29K used.
Second level of WORKING-STORAGE-AREA: capacity 98K, 2048K allocated, 47K used.

Status of the dynamic garbage collector:                      On
Dynamic (new+copy) space 6,490,761.  Old space 0.  Static space 12,479,751.
Free space 26,574,848.  Committed guess 22,488,118, leaving 3,824,586 to use
 before flipping.
There are 9,779,900 words available before Start GC :Immediately might run out
 of space.
Doing Start GC :Immediately now would take roughly 33 minutes.
There are 26,574,848 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 328 (1 full GC, 2 dynamic GC's, and 325 ephemeral
 GC's).
Since cold boot 53,043,930 words have been consed, 45,867,153 words of garbage
 have
been reclaimed, and 11,658,295 words of non-garbage have been transported.
The total "scavenger work" required to accomplish this was 121,864,225 units.
Use Set GC Options to examine or modify the GC parameters.
```

The command Start GC turns on the garbage collector.

Start GC *keywords*

Controls the operation of the Garbage Collector. Start GC with no keywords turns on both dynamic and ephemeral garbage collection.

keywords :Cleanup, :Dynamic, :Ephemeral, :Immediately, :Selective

:Cleanup {Yes, No, Ask} Whether or not to run GC Cleanups to attempt to free address space. The default is No. The mentioned default is Yes, which does the maximum cleanup possible. Ask queries you about each cleanup before performing it. Start GC :Cleanup without other keywords does not perform a GC or alter the mode of the background GC. See the section "GC Cleanups".

```
Command: :Start GC :Cleanup (Yes, No, or Ask [default Y
GC Cleanup Tasks
  Reset all input histories?  Yes  No
  Reset all presentation histories?  Yes  No
  Reset all editor histories?  Yes  No
  Reset LISP-TOP-LEVEL variables such as '* and '+?  Ye
  Reset interactor output histories?  Yes  No
  Clear some resources?  Yes  No
<ABORT> aborts, <END> uses these values
```

:Dynamic {Yes, No} Enables or disables the dynamic level of incremental GC.

:Ephemeral {Yes, No} Enables or disables the ephemeral level of incremental GC.

:Immediately {Yes, No, In-Place, By-Area} Performs a complete garbage collection right now. The mentioned default is Yes. In-Place garbage collection does not copy objects; rather, it compacts good objects to the bottom of each region. Since there are never two copies of an object during the garbage collection, the virtual memory (paging space) required for GC is reduced. It is slower and non-interruptable, and thus is recommended only when execution speed and interaction are not important and when there is insufficient disk space for normal garbage collection.

Start GC :Immediately By-Area is the same as Start GC :Immediately Yes :Selective Yes.

Start GC :Immediately also offers to run GC Cleanups.

:Selective {Yes, No} Specifies areas in which to collect garbage. When your address space has shrunk to where there is not quite enough free space for Start GC :Immediately to complete, :Selective suggests some areas to flip that will maximize the amount of space reclaimed without risking running out of space completely. :Selective can be used with both Start GC :Immediately Yes and Start GC :Immediately In-Place.

For more information about the process of Garbage Collection, see "Theory of Operation of the GC Facilities" and "Invoking the Garbage Collection Facilities".

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".

Compressing Data

It is possible to compress and decompress data to save space on disk and to interoperate with files compressed on UNIX systems. There are two CP commands to do this, Compress File and Decompress File.

Compress File Command

Compress File *input-files output-files keywords*

Compresses the data in *input-files* and produces *output-files*. Wildcards are allowed. If *input-files* and *output-files* are the same files, the input files are replaced by the output files.

input-files {*pathname(s)*} One or more files to compress.

output-files {*pathname(s)*} One or more files to contain the compressed data.

keywords :Copy Properties, :Create Directories, :More Processing, :Output Destination :Preamble Type, :Query, :Translation Strategy

:Copy Properties {*list of file properties*} The properties you want duplicated in the new files. The default is author and creation date.

:Create Directories
{Yes, Error, Query} What to do if the destination directory does not exist. The default is Query.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Preamble Type {Symbolics, UNIX} Type of preamble to use.

:Query {Yes, No, Ask} Whether to ask before compressing each file.

:Translation Strategy
{Text, Binary, Query, Heuristicate} Whether or not to perform character set translation. Text means to do ASCII character set translation, reading each input file as a text file. Binary means not to do ASCII character set translation, reading each input file as a binary file. Query asks, for each file, whether to treat the file as text or binary. Heuristicate attempts to guess whether the file is text or binary based on its name, as follows: The filename is broken up into *words*, where each word is separated by a non-alphanumeric character. A rightmost word of "Z" is removed (if present). Then the current rightmost word is checked against **compression::*likely-unix-binary-formats*** If a match is found, the file is assumed to be binary, otherwise it is assumed to be text.

:Translation Strategy is only useful if you are reading or writing a file with a UNIX-style compression preamble, because Symbolics-style compression preambles record the element type and character set of the compressed data. Using :Translation Strategy with a file having a Symbolics-style compression preamble is ignored with a warning.

Decompress File Command

Decompress File *input-files output-files keywords*

Decompresses the data in *input-files* and produces *output-files*. Wildcards are allowed. If *input-files* and *output-files* are the same files, the input files are replaced by the output files.

input-files {*pathname(s)*} One or more files to decompress.

output-files {*pathname(s)*} One or more files to contain the decompressed data.

keywords :Copy Properties, :Create Directories, :More Processing, :Output Destination :Preamble Type, :Query, :Translation Strategy

:Copy Properties {*list of file properties*} The properties you want duplicated in the new files. The default is author and creation-date.

:Create Directories
{Yes, Error, Query} What to do if the destination directory does not exist. The default is Query.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this

command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Query {Yes, No, Ask} Whether to ask before decompressing each file.

:Translation Strategy
{Text, Binary, Query, Heuristicate} Whether or not to perform character set translation. Text means to do ASCII character set translation, writing each resulting file as a text file. Binary means not to do ASCII character set translation, writing each resulting file as a binary file. Query asks, for each file, whether to treat the file as text or binary. Heuristicate attempts to guess whether the file is text or binary based on its name, as follows: The filename is broken up into *words*, where each word is separated by a non-alphanumeric character. A rightmost word of "Z" is removed (if present). Then the current rightmost word is checked against **compression::*likely-unix-binary-formats***, and if a match is found, the file is assumed to be binary, else it is assumed to be text.

:Translation Strategy is only useful if you are reading or writing a file with a UNIX-style compression preamble, because Symbolics-style compression preambles record the element type and character set of the compressed data. Using :Translation Strategy with a file having a Symbolics-style compression preamble is ignored with a warning.

Programmers wishing to use data compression in their applications can use the Compression Substrate. See the section "Compressing Data - the Compression Substrate".

Understanding Character Styles

See the section "Using Character Styles in Zmacs".

What is a Character Style?

A *character style* is a combination of three characteristics that describe how a character appears. These characteristics are the *family*, *face*, and *size*.

Family Characters of the same family have a typographic integrity, so that all characters of the same family resemble one another. Examples: SWISS, DUTCH, and FIX.

Face	A modification of the family, such as BOLD or ITALIC.
Size	The size of the character, such as NORMAL or VERY-SMALL.

The character style is the grouping of the family, face, and size fields. A character style is often represented by the convention:

family.face.size

An example of a fully specified character style is:

```
SWISS.ITALIC.LARGE
```

Each element of the character style can be specified or left unspecified. A family, face, or size of NIL means to use the default value. Most characters have the following character style:

```
NIL.NIL.NIL
```

Characters of style NIL.NIL.NIL are displayed in the default character style established for the current output device.

Default Character Styles

The appearance of a character depends on two things: the character style of the character, and the default character style. The default character style is a global parameter of an output device. It applies for all processes. Windows, buffers, files, and printers each have default character styles for output. The default character style specifies the appearance of a character whose character style is NIL.NIL.NIL. The character's style is merged against the default character style to produce the final appearance of the character. A default character style must be fully specified.

We recommend that you use character styles by making good use of the default character styles. You preserve the most flexibility by keeping the character style of the characters themselves as unspecified as possible. If you want to change the appearance of all characters in a Zmacs buffer, a Zmail message or a window, you can change the default character style instead of changing the character style of each character.

The default character style affects the appearance of a character on output. There is also a typein character style for each interactive stream, which is normally NIL.NIL.NIL. The typein character style affects the character style in which characters are entered as input. If the typein character style is NIL.BOLD.NIL, any characters you enter at the keyboard have the character style NIL.BOLD.NIL. It is important to be sure that the application program can handle characters whose character style is something other than NIL.NIL.NIL, if you are going to use a typein character style other than NIL.NIL.NIL.

If you only want to change the way that characters echo, but not the way they are entered as input, you can change the echo character style. See the section "Using Character Styles in the Input Editor".

Merging Character Styles

This section gives some examples of how the character style of a character is merged against the default character style to produce a final result.

In general, we advise that you specify as little as possible when changing a character style. That is, if you want the character's face to be italic, specify only the face component and let the family and size come from the default character style.

Character Style of a Character	*Default Character Style*	*Result of Merging*
NIL.NIL.NIL	FIX.ROMAN.NORMAL	FIX.ROMAN.NORMAL
NIL.ITALIC.LARGE	FIX.ROMAN.NORMAL	FIX.ITALIC.LARGE
NIL.ITALIC.SMALLER	FIX.ROMAN.NORMAL	FIX.ITALIC.SMALL
SWISS.BOLD.LARGER	FIX.ROMAN.NORMAL	SWISS.BOLD.LARGE
SWISS.BOLD.SAME	FIX.ROMAN.NORMAL	SWISS.BOLD.NORMAL

The family and face components are either NIL or the name of a family or face.

The size component can be NIL, an *absolute size* (such as LARGE or VERY-SMALL) or a *relative size* (such as LARGER or SMALLER). A relative size is merged against the default size such that when you merge LARGER against NORMAL, the result is the next size larger than NORMAL.

The ordered hierarchy of sizes is:

TINY
VERY-SMALL
SMALL
NORMAL
LARGE
VERY-LARGE
HUGE

If you try to merge SMALLER against the smallest size, TINY, the result is TINY. Similarly, if you try to merge LARGER against the largest size, HUGE, the result is HUGE.

Using Character Styles in the Input Editor

The default character style for the input editor is FIX.ROMAN.NORMAL.

You can use the Set Window Options CP command to change the default character style, the typein character style, or the echo character style. The default character style and typein character style are described elsewhere. See the section "Default Character Styles".

The echo character style affects the way that characters you enter at the keyboard are echoed. The appearance of characters that you type depends on: the character style of the character (which is usually the same as the typein style), which is merged against the echo character style, which is merged against the default character style.

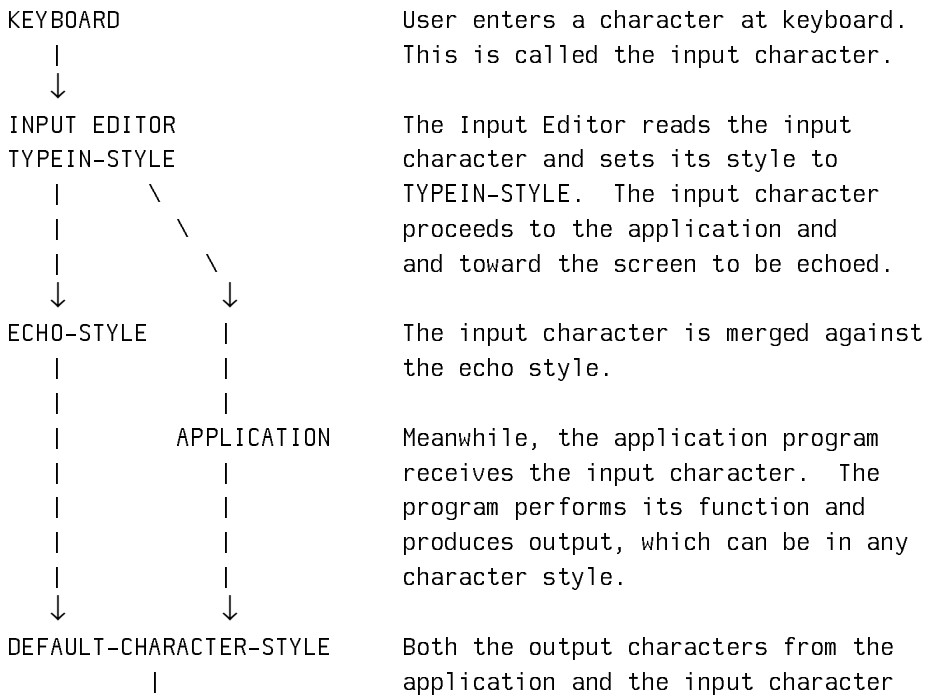
It is important to be sure that the application program can handle characters whose character style is something other than NIL.NIL.NIL, if you are going to use a typein character style other than NIL.NIL.NIL. If you only want to change the way that characters echo, but not the way they are entered as input, you can change the echo character style. See the section "Character Styles and the Lisp Listener".

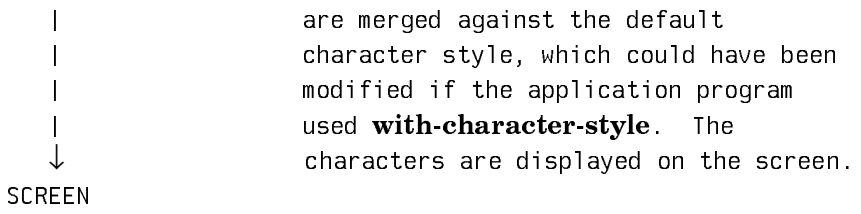
In addition to the Set Window Options command, you can change the typein character style in the input editor by using c-m-J. You are then prompted for a character style. Enter something in the *family.face.size* convention, such as DUTCH.BOLD-ITALIC.LARGER.

Character Styles and the Lisp Listener

This section and diagram describes the role of the typein character style, the echo character style, and the default character style in the Lisp Listener.

When you type a character at the keyboard, it follows one path which eventually causes it to be echoed on the screen. The same character follows a path to the application program. The application program might produce some output, which is also displayed on the screen.





For example:

Typein style is: SWISS.NIL.NIL
Echo style is: NIL.BOLD.NIL
Default character style is: FIX.ROMAN.NORMAL

The input editor reads in the input character according to the typein style, so the input character has the character style of SWISS.NIL.NIL. The input character is merged against the echo style, so it is then SWISS.BOLD.NIL. The input character is then merged against the default character style, so it is finally displayed on the screen in the character style SWISS.BOLD.NORMAL.

Meanwhile the original input character of style SWISS.NIL.NIL is sent to the application. As it runs, the application program produces output characters of style NIL.BOLD-ITALIC.NIL. The output characters are merged against the default character style; they are displayed in the character style FIX.BOLD-ITALIC.NORMAL.

Note that the application program can use **with-character-style**, **with-character-face** and so on when producing output. If this is done, the specified style information is merged against the default character style. Thus it affects both the way that the input characters are echoed and the way any output characters are displayed.

If you want to specify how your input characters appear, you can change the echo style. If you want your input characters to have the character style set to something other than NIL.NIL.NIL, you can change the typein style.

Using Character Styles in Zmail

Every message has its own default character style used for displaying the message. The default is recorded in a new **Default-Character-Style** header. If this header is not present, the message is displayed using FIX.ROMAN.NORMAL as the default character style.

You can use the command Set Message Default Character Style (m-X) to change the default character style of the current message (if you are examining a message), or the message you are composing. When you are prompted for a character style, you must enter a complete character style; it cannot contain NILs. This command is also available in the Editor menu, which is offered when you click Right.

When you are in Zmail, you can use the Zmacs commands for changing character styles. For a list of available commands and a description of how to enter a new character style, see the section "Using Character Styles in Zmacs".

Using Character Styles in Hardcopy

When you hardcopy a file, the outcome depends on the character style of the characters in the file and the printer's default values for the body character style and header character style.

The printer's defaults are stored in the printer object in the namespace database, in the attributes:

body-character-style
: The default character style to be used by this printer

headers-character-style
: The default character style to be used for the headers by this printer.

You can override the defaults stored in the printer objects by using **setq** on the variable **hardcopy:*hardcopy-default-character-styles***. Note that the value of **hardcopy:*hardcopy-default-character-styles*** is merged with the default style for the printer. See the variable **hardcopy:*hardcopy-default-character-styles***.

Understanding Networks and the Namespace System

Introduction to the Namespace Database

A namespace database contains objects. An object must belong to one (and only one) of seven classes in order to be registered in the namespace database:

Host	Represents any computer, usually connected to a network.
User	Represents a person who uses any of the hosts, or a daemon user.
Network	Represents a computer network, to which some hosts are attached.
Printer	Represents a device for producing hardcopy.
Site	Represents a collection of hosts, printers, and networks, grouped together in one location.
Namespace	A database containing information about the mappings from object names to objects, and about the objects themselves.
File System	Reserved for Statice.

Objects in the namespace database have global-names (which identify them), and attributes (which describe them).

Data Types for Attributes

The following data types can be used with attributes in the namespace database:

Global-Name	A name that is shared by all namespaces.
Token	An arbitrary character string.
Set	One or more elements of the same data type.
Pair	Two elements; each element can be of a different data type.
Triple	Three elements; each element can be of a different data type.

Global-name and *token* require you to supply one value. *Set*, *pair* and *triple* require you to supply compound (one or more, two, or three) values. For more information about attributes, see the section "Attributes for Objects in the Namespace Database".

Use the Namespace Editor to "create" (or document) objects, their global-names, and their attributes. For more information about the Namespace Editor, see the section "Using the Namespace Editor".

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. An object is identified both by its class and its name.

This means that 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". You cannot say "Find George".

A namespace is a database that contains mappings from names to objects. Names in one namespace are unrelated to names in another namespace. Specifically, a namespace maps from [class, name] pairs to objects, since every object is identified both by its class and by its name.

Typical sites have one namespace, and the names of all the objects at that site are in that namespace. An object in some namespace other than your own can be referred to by a *qualified name*, which consists of the name of the namespace, a vertical bar, and the name of the object in that namespace.

When long-distance networks link together different sites, the possibility for name conflicts arises. Neither site is forced to change its names just because it wants to connect to the other site.

For example, suppose both Harvard and Yale have computer centers. Harvard has three hosts named Yellow, Orange, and Blue. Yale has three hosts named Apple, Orange, and Banana. Each computer center has its own namespace; Harvard's is named Harvard and Yale's is named Yale.

At Harvard, the Harvard computers are referred to by their unqualified names (Yellow, Orange, and Blue), but the Yale computers are referred to by their 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 for its *search rules*. When a name is looked up, each of the *search rules* namespaces listed is consulted in turn, until an object of the desired name is found in one of them. If you list namespaces other than your own in your *search rules*, it is easier to refer to objects in those namespaces, because you do not need to use qualified names for them (unless a name conflict exists).

For example, in the scenario above, the *search rules* for the Harvard namespace could list 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 Yale|Orange is only necessary because a name conflict exists.

Actually, only some classes of objects have names that are in namespaces; other classes of objects are *globally* named, which means that their names are universal, and conflicts are not permitted. In particular, namespaces and sites are globally named; networks, hosts, printers, and users are not (instead, they're named within namespaces).

Some namespaces do not correspond to any local site. Most large nationwide or worldwide networks have their own host-naming convention. For example, the U.S. Department of Defense Internet has its own set of host names, and this is considered a namespace. If a local site includes some hosts that are on the Internet, it might want to put the Internet namespace into its search list, and install gateways to access Internet machines. For more information, see the section "Namespace Editor CP Commands".

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 for the object. Sometimes you give an object a nickname because its full name is too long to type conveniently. However, each object only has one primary name.

It is possible for an object to be in several namespaces at once. For example, a host which is on both the Internet and a local network at some site might be in both the Internet namespace and the local namespace. In this case, both namespaces maintain their 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 followed.

Using the Namespace Editor

You use the Namespace Editor to register new users and new hardware in the namespace database. To do so, you create and save namespace objects representing the new addition to the site. Once an object has been globally saved, it becomes part of your site's configuration.

The Namespace Editor checks input, and supplies both help and completion. Specifically, the Namespace Editor:

- Checks for errors in network addresses.
- Verifies the nicknames in use by other hosts in the local namespace.
- Checks for unknown services, mediums, and protocols.

You can reach the Namespace Editor in these ways:

- Use the Command Processor (CP) Select Activity command (and select the Namespace Editor Activity). See the section "Workbook: Selecting a New Activity".
- Assign the Namespace Editor to a SELECT key combination with the Select Key Selector or the **tv:add-select-key** function. For more information about how to use the SELECT key, see the section "Customizing the SELECT Key".
- Invoke individual Command Processor (CP) Namespace Editor commands in a Lisp Listener. See the section "Namespace Editor CP Commands".

Creating a New Namespace Object

First select the Namespace editor by using the Edit Namespace Object command. To create a new namespace object, click on [Create Object]. You are prompted for the class and the name of the new object. A template for the information is displayed in the top window. The attributes are mouse sensitive. Clicking on an attribute prompts you in the bottom window for the information to put in the attribute.

Note that the required attributes appear with an asterisk (*) after them. All object classes have a small number of required attributes, and several optional attributes.

You can also create a new object by copying an existing object. Enter Copy Object at the Namespace Editor prompt. Alternatively, use Create Object with the :Copy From keyword.

The window can be scrolled using the SCROLL and m-SCROLL keys or with the mouse. See the section "Scrolling with the Mouse".

When you are satisfied with the information, you can enter it in the database by clicking on [Save Object]. Then click on [Quit] to exit the namespace editor.

For a discussion of saving (locally or globally) new information in the namespace database: See the section "Editing a Namespace Object".

Editing a Namespace Object

Select the Namespace Editor by using the Edit Namespace Object command. If you do not supply the class and object name to Edit Namespace Object, the Namespace Editor window comes up empty and you can click on [Edit Object] or enter the Edit Object command. You are prompted for the name of an object to edit. The current information for the object is retrieved from the namespace database and displayed in the window.

Click Middle on the attribute name for information on the attribute.

The attribute fields are mouse-sensitive. Clicking on an attribute prompts you for information. Mouse clicks have the following meaning:

Left Replace the information in the attribute.

Middle Edit information in the attribute.

Right Menu.

sh-Middle Delete the information in the attribute.

The window can be scrolled using the SCROLL and m-SCROLL keys, the scroll bar, or the mouse.

Once you have finished editing the information, you have three ways to proceed. You can click on [Quit] without saving the changed information. If you are just practicing using the Namespace Editor, that would be appropriate.

The other two choices are to save the information locally or globally. If you save it globally, the new information is stored in the site's namespace database. If you save it locally, the new information is stored only in your machine's local copy of the namespace; changes saved locally affect only your machine (until it is rebooted).

The initial state of the Namespace Editor is the global mode. When you are in global mode the middle of the screen looks like:

Editing HOST TOWHEE in namespace SCRC

If you have clicked on [Locally], you are in local mode. The middle of the screen looks like:

Editing HOST TOWHEE (locally) in namespace SCRC

You can click on [Locally] to toggle the mode between global and local. When you are ready, click on [Save Object] to save the information. Then click on [Quit] to exit the Namespace Editor.

For a complete list of the namespace editor commands, see the section "Namespace Editor Commands".

Connecting to a Remote Host over the Network

If your Symbolics computer is on a network and configured properly, you can access other hosts on the network with the Terminal program.

To use the Terminal program, press SELECT T. The prompt is:

```
Connect to host:
```

Type the name of the host to which you want to connect. The network system makes a connection, and you will see the prompt of the remote host displayed on the screen. You are now communicating directly with the remote machine.

When you are connected to a remote host, the NETWORK key provides several useful commands. For example:

NETWORK HELP	Displays the list of options for the NETWORK key.
NETWORK L	Logs out of the remote host, and breaks the connection.
NETWORK D	Disconnects without logging out first.

See the section "NETWORK Key".

If you want to use the Terminal program to log in to a remote Symbolics computer when someone is logged in to that machine, you must first enable remote login by evaluating the form **(net:remote-login-on)** on that machine. See the function **net:remote-login-on**.

See the section "Connection Keywords in the Terminal Program". See the section "Dynamic Window Features of The Terminal Program".

Remote Terminal Commands

Set Remote Terminal Options
: Enables you to toggle MORE processing on and off. Additionally, you can specify whether a status line appears at the bottom of your screen, and also how often the status line updates.

Show Remote Terminal Options
: Enables you to view the current settings of your remote terminal options. This command also displays the height and width of your screen.

Halt Remote Terminal
: Enables you to halt your remote terminal.

A Brief Introduction to the Lisp World

Documentation Notation Conventions for Lisp Objects

Functions

A typical description of a Lisp function looks like this:

function-name *arg1 arg2* &optional *arg3 (arg4* **(foo3)***)* *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** 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 of an argument 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".) 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:

"Typein" or "example" font (`function-name`)

Indicates something you are expected to type. This font is also used for Lisp examples that are set off from the text and in some cases for information, such as a prompt, that appears on the screen.

"Key" font (`RETURN`, `c-L`)

For keystrokes mentioned in running text.

Macros and Special Forms

The descriptions of special forms and macros look like the descriptions of these imaginary ones:

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.
- 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).
- Braces 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 pa-

rameters. 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.

Flavors, Flavor Operations, and Init Options

Flavors themselves are documented by the name of the flavor.

Flavor operations are described in three ways: as methods, as generic functions, and as messages. When it is important to show the exact flavor for which the method is defined, methods are described by their function specs. Init options are documented by the function spec of the method.

When a method is implemented for a set of flavors (such as all streams), it is documented by the name of message or generic function it implements.

The following examples are taken from the documentation.

sys:network-error *Flavor*

This set includes errors signalled by networks. These are generic network errors that are used uniformly for any supported networks. This flavor is built on **error**.

(flavor:method :clear-window tv:sheet) *Method*

Erases the whole window and move the cursor position to the upper left corner of the window.

:tyo *char* *Message*

Puts the *char* into the stream. For example, if **s** is bound to a stream, then the following form will output a "B" to the stream:

```
(send s :tyo #\B)
```

For binary output streams, the argument is a nonnegative number rather than specifically a character.

dbg:special-command-p *condition special-command* *Function*

Returns **t** if *command-type* is a valid Debugger special command for this condition object; otherwise, returns **nil**.

The compatible message for **dbg:special-command-p** is:

:special-command-p

For a table of related items, see the section "Basic Condition Methods and Init Options".

(flavor:method :bottom tv:sheet) *bottom-edge* *Init Option*

Specifies the y-coordinate of the bottom edge of the window.

Variables

Descriptions of variables ("special" or "global" variables) look like this:

typical-variable *Variable*

The variable ***typical-variable*** has a typical value....

The Lisp Top Level

These functions constitute the Lisp top level and its associated functions.

si:lisp-top-level *Function*

Initializes the Lisp environment and then starts the initial Lisp Listener process running **si:lisp-top-level1**.

si:lisp-top-level1 &optional *(stream* **zl:terminal-io***)* *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. The values of *, +, -, /, ++, **, +++, and *** are maintained.

prin1 *Variable*

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

Some Utility Functions

All of these Lisp functions are available as Command Processor commands. However, it is useful to know them should you ever find yourself in a situation where you cannot use the Command Processor. **zwei:save-all-files**, for example, is useful if you are in the cold load stream and cannot get out without booting. See the section "The Cold Load Stream".

zwei:save-all-files &optional (*ask* **t**) *Function*

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 the editor. 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.

In CLOE runtime, allows the user to enter an editor. If *arg* is a pathname, the associated file is edited; if *arg* is a symbol, the associated function definition is edited. CLOE first checks the value of **cloe::*editor-for-ed*** and then checks the value of the shell environment variable CLOED. If either of these supply a string to use as the editor invoking command, it is used, otherwise "ed" is used.

zl: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).

zl:mail &optional *initial-destination initial-body prompt initial-idx bug-report (make-subject* **(memq zwei:*require-subjects* '(t :init))***) initial-subject* *Function*

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 **zl:mail** function.

prompt and *initial-idx* are used by programs, such as **zl:bug**, that call **zl:mail**. *prompt* is a string printed in the minibuffer of the mail window created by **zl:mail**. *initial-idx* positions point in the mail window.

zl:bug &optional *(system* **(or dbg:*default-bug-report-recipient-system* dbg:*default-default-bug-report-recipient-system*)**) *additional-body prompt point-before-additional-body (make-subject* **(memq zwei:*require-subjects* '(t :init :bug))**) *initial-subject* *Function*

Reports a bug. This is the same as the Report Bug command. **zl:bug** is like **zl:mail** but includes information about the system version and your machine in the text of the message.

system is the name of the faulty program (a symbol or a string). It defaults to **Genera** (the software 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.

You can control the character style of the system information inserted at the beginning of the message by setting the value of **dbg:*character-style-for-bug-mail-prologue***. See the variable **dbg:*character-style-for-bug-mail-prologue***.

zl: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 give just a name, **zl:qsend** looks at all the Symbolics machines at your site to find any that *name* is logged into. If the user is logged in to one Symbolics machine, it is used as the host; if more than one, **zl:qsend** asks you which one you mean. If you leave out *destination* altogether, doing just **(zl:qsend)**, Converse is selected as if you had pressed SELECT C.

message should be a string. For example:

```
(qsend kjones@wombat "Want to go to lunch?")
```

If *message* is omitted, **zl: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 **zl:qsend**. So you get some editing power, although not as much as with full Converse (since the latter uses

Zwei). See the section "Editing Your Input". **zl:qsend** predates Converse and is retained for compatibility.

Checking on What the Machine is Doing

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. Many of these commands have Command Processor equivalents, but the Lisp versions of them are often useful in situations where you cannot use the Command Processor, for instance when you are in the Cold Load Stream. They are useful for finding information about your current state or about the Lisp world in general.

Functions for Examining Objects in the Lisp World

who-calls *symbol* &optional *how*
: Tries to find all the functions in the Lisp world that call *symbol*. See the "Show Callers Command".

si:who-calls-unbound-functions
: Searches the compiled code for any calls through a symbol that is not currently defined as a function.

what-files-call *symbol-or-symbols* &optional *how*
: Returns a list of the pathnames of all the files that contain functions that **who-calls** would have printed out.

apropos *string* &optional *package (do-inherited-symbols* **t***) do-packages-used-by*
: Searches for all symbols whose print-names contain *string* as a substring. See the "Find Symbol Command".

where-is *pname*
: Finds all symbols named *pname* and prints a description of each symbol.

describe *anything* &optional *si:*describe-no-complaints**
: Provides all the interesting information about any object (except array contents).

inspect &optional *object (frame* **tv:*default-inspector-frame****)*
: A window-oriented version of **describe**. See the "Inspect Command".

disassemble *function* &optional *from-pc to-pc*
: Prints out a human-readable version of the macroinstructions in *function*. See the "Show Source Code Command".

Variables for Examining the Lisp World

These variables are particularly useful for typing Lisp interactively, especially for examining your state in the Debugger.

-	Holds the form being evaluated.
+	Holds the previously evaluated form.
++	Holds the form evaluated two interactions ago.
+++	Holds the form evaluated three interactions ago.
*	Holds the result of the previous evaluation.
**	Holds the result of the form evaluated two interactions ago.
***	Holds the result of the form evaluated three interactions ago.
/	Holds a list of the results of the previous evaluation.
//	Holds a list of results from two interactions ago.
///	Holds a list of the results from three interactions ago.

Debugging Functions and Variables

grindef &rest *fcns* Prints the definitions of one or more functions, with indentation to make the code readable.

break &optional *format-string* &rest *format-args*
Enters the Debugger. Equivalent to `c-m-SUSPEND`.

sys:*break-bindings*
Holds a list of variables and values to which to bind the variables when a breakpoint is encountered.

dbg:*debugger-bindings*
Holds a list of variables and values to which to bind the variables when the Debugger is encountered.

Utility Functions

The variable ***features*** holds the list of features included in Genera.

features *Variable*

Returns a list of symbols indicating features of the Lisp environment. The default list for Genera is:

```
(:DEFSTORAGE :DEBUG-SCHEDULER-QUEUES :NEW-SCHEDULER :LOOP
:DEFSTRUCT :LISPM :SYMBOLICS :GENERA :ROW-MAJOR machine-type
:CHAOS :IEEE-FLOATING-POINT :SORT :FASLOAD :STRING :NEWIO
:ROMAN :TRACE :GRINDEF :GRIND)
```

The value of this list is kept up to date as features are added or removed from the Genera system. Most important is the symbol *machine-type*; this is either **3600** or **:imach** and indicates on which type of Symbolics machine the program is running. The order of this list should not be depended on, and might not be the same as shown above.

Features SYMBOLICS and CLOE are present in both the CLOE Developer and the CLOE Application Generator. Feature CLOE-DEVELOPER is present only in the CLOE Developer, and feature CLOE-RUNTIME is present only in the Application Generator.

```
*features* =>
(:CLOE-RUNTIME :LOOP :INTEL-386 :UNIX-V3 :CLOE :IEEE-FLOATING-POINT
:SYMBOLICS)
```

The following functions return specifics about your machine and your environment.

lisp-implementation-type *Function*

Returns a string that is the name of the Lisp system running on your machine.

```
(lisp-implementation-type) => "Symbolics Common Lisp"
```

or

```
(lisp-implementation-type) => "Symbolics CLOE"
```

lisp-implementation-version *Function*

Returns a string that identifies the current version of the system running on your machine, including the patch level and microcode.

```
(lisp-implementation-version)
 => "System 424.207 3640-MIC microcode 428"
```

For the CLOE Developer,

```
(lisp-implementation-version)
=>"1.1, Cloe Developer 318.0"
```

and for the CLOE Application Generator,

```
=>(lisp-implementation-version)
"CLOE Application Generator 1.1"
```

machine-type *Function*

Returns a string that identifies the kind of hardware you are using.

```
(machine-type) => "Symbolics 3620"
```

For the CLOE Developer,

```
(machine-type)
=>"Symbolics"
```

and for the CLOE Application Generator,

```
(machine-type)
=>"Intel"
```

machine-version *Function*

Under Genera, returns the board-level hardware information about your machine. This is the same as the information displayed by the Show Machine Configuration command for your machine.

Under CLOE, returns a string indicating the current version of the machine for current implementation. For example, for the CLOE Developer you might get something like the following:

```
(machine-version)
=>"3640"
```

and for the CLOE Application Generator

```
(machine-version)
=>"386"
```

machine-instance *Function*

Returns a string that is the name of your machine.

```
(machine-instance) => "WOMBAT"
```

This is the contents of the `Host` field in your machine's namespace object. See the section "Why do you name machines and printers?".

software-type *Function*

Returns a string that is the name of the operating system Lisp is running in.

```
(software-type) => "Lisp Machine"
```

For the CLOE Developer

```
(software-type) =>"Genera"
```

and for the CLOE Application Generator

```
(software-type) =>"UNIX" or "MS-DOS"
```

software-version *Function*

Returns a string that represents the versions of all the systems running in your world. This includes any local systems you have loaded. This is similar to the information displayed by the Show Herald command.

For the CLOE Developer

```
(software-version) =>"8.0"
```

and for the CLOE Application Generator

```
(software-version) =>"V.3" or "3.1"
```

short-site-name *Function*

Returns a string that is the name of your site. This is the contents of the `Site` field in your site's namespace object.

The CLOE Runtime environment does not provide a uniform way to obtain a "site" designation. If the value of the variable **cloe::*short-site-name*** is **nil**, you are prompted to enter the correct values for your site. Initially, **cloe::*short-site-name*** is set to `"CLOE-USER-SITE."`

long-site-name *Function*

Returns a string that is the full name of your site. This is the contents of the `Pretty-name` field in your site's namespace object.

The CLOE Runtime environment does not provide a uniform way to obtain a "site" designation. If the value of the variable **cloe::*long-site-name*** is **nil**, you are prompted to enter the correct values for your site. Initially, **cloe::*long-site-name*** is set to `"CLOE-USER-SITE"`.

print-sends &optional (*stream* **zl: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.

zl:print-notifications &optional *(from* **0***) (to* **(1- (zl: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 Genera itself. If *from* or *to* is specified, prints only part of the notifications list.

Example: `(zl:print-notifications 0 4)` prints the five most recent notifications.

This is the same as the "Show Notifications Command".

si:show-login-history &optional *(whole-history* **si:login-history***)* *Function*

Prints one line for each time the login command has been used since the world was last cold booted. See the section "Show Login History Command".

zl:hostat &rest *hosts* *Function*

Asks each of the *hosts* for its status, and prints the results. If no hosts are specified, asks all hosts on the Chaosnet. 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 Symbolics host 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.

See the section "Show Hosts Command".

To abort the host status report produced by **zl:hostat** or `FUNCTION H`, press `c-ABORT`.

net: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. **net:uptime** prints out the results. If **net: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. Note that if you do not specify a host, this command returns the status of every host on the network.

The **net:uptime** function is a variant of **zl:hostat**.

Dribble Files

There are a number of ways to capture output. You can use the Command Processor command Copy Output History, or you can use the :Output Destination keyword with some operation. Alternatively, you can do it programmatically, using dribble files. 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 &optional *pathname editor-p*
Opens *pathname* as a "dribble file".

(dribble) Closes the file.

zl:dribble-start *pathname* &optional *editor-p (concatenate-p* **t***) debugger-p*
Opens *pathname* as a "dribble file".

zl:dribble-end Closes the file opened by **zl:dribble-start** and resets the I/O streams.

Using Peek

Overview of 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 interval between updates is 20 seconds, but if you want more or less frequent updates, you can set it 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. SCROLL or C-V scrolls the window forward (towards the bottom), M-SCROLL or M-V scrolls it backward (towards the top).

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.

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 to Processes".

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".

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, Zwei sectionization, netboot and the disk. There are two types of 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 "Theory of Operation of the GC Facilities".

File System (F)

File System mode, invoked by pressing F or by clicking on [File System], provides 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 or pressing HELP displays the help information that is displayed when Peek is selected the first time.

Clicking on [Quit] or pressing Q buries the Peek window and returns you to the window from which you invoked Peek.

Tools for Lisp Debugging

Here is an introduction to various tools for debugging, understanding, or improving Lisp programs. In addition to the Debugger, this section introduces the Flavor Examiner and the Inspector.

Overview of the Debugger

Genera, the Symbolics software environment, offers you a host of powerful debugging tools. The most comprehensive of these tools is the Symbolics interactive Debugger and its window-oriented counterpart, the Display Debugger.

Other debugging tools are:

- The *Trace* facility, which performs certain debugging actions when a function is called or when a function returns. See the section "Tracing Function Execution".

- The *Advise* facility, which modifies the behavior of a function. See the section "Advising a Function".

- The *Step* facility, which allows you to execute interpreted forms in your program, one at a time, so that you can examine what is happening when execution suspends at every "step." See the section "Stepping Through an Evaluation". The Debugger's :Single Step command also performs stepping through compiled functions. See the section "Single Step Command".

- The *evalhook* facility, which allows you to get a particular Lisp form whenever the evaluator is called. The Step facility also uses **evalhook**. See the section "A Hook Into the Evaluator".

- The *Inspector* is a window-oriented program that lets you inspect data objects and their components. See the section "The Inspector".

- *Peek* is a program that gives a dynamic display of various kinds of system status. See the section "Using Peek".

- The *Metering Interface* allows you to meter the performance of a form, function, or process. See the section "Metering Interface".

For information on the Display Debugger, see the section "Using the Display Debugger".

In the Genera software environment, unlike more traditional programming environments, you do not have to include the Debugger explicitly when you compile your programs. Generally, you can debug your code as you write it without having to perform a series of complicated compiling, loading, and executing procedures between source code development and debugging.

Because Symbolics user-interface features allow you to perform many Symbolics activities simultaneously — Zmacs, Zmail, the file system, a Dynamic Lisp Listener, and so on — debugging becomes an easy task, regardless of how many system activities you are using. You can move in and out of the Debugger as easily as you can move in and out of any other activity in Genera.

For example, the Debugger command `c-E` (:Edit Function) brings up a specified function for you to edit in a Zmacs editor window. This is useful when you have found the function that caused the error and want to edit that function immediately. Another command, `c-M` (:Mail Bug Report), creates a bug report message in a mail window and puts a backtrace into it. While composing the bug report, you can switch back and forth between the Debugger and the mail window.

The Debugger is there whenever you need it. It is invoked whenever an error occurs in your program's execution or the execution of a system function. That is, your machine brings you into the Debugger whenever it encounters an error that is not handled by a condition handler, for example, when you reference an unbound variable. See the section "Entering and Exiting the Debugger". Once in the Debugger, you are given a choice of actions that can correct the error. These actions are called *proceed* and *restart options*. See the section "Proceeding and Restarting in the Debugger".

You can also enter the Debugger explicitly, at any time, by pressing `m-SUSPEND` or `c-m-SUSPEND`. Or you can make your program enter the Debugger by inserting the **break** or **zl:dbg** function into your program code. See the section "Entering and Exiting the Debugger".

Upon Debugger entry, besides selecting one of the proceed and restart options, you can enter any of the Debugger's commands. These commands are full-form English commands, built on the normal Command Processor (CP) substrate. In fact, several Debugger commands are in the global command table. For more information on Debugger commands, see the section "Entering a Debugger Command" and see the section "Debugger Command Descriptions".

In the Debugger you can also evaluate a form in the lexical (user-program) context of the current frame. This context is referred to as the Debugger's *evaluation environment*. You can think of the Debugger's evaluation environment as a special

read-eval-print loop that not only evaluates forms but also evaluates them in the context of the suspended function, where the lexically apparent values of all the local variables are accessible. For more information on the evaluation environment, see the section "Evaluating a Form in the Debugger".

Like other output in the Genera software environment, Debugger output is mouse sensitive, so you can perform many useful Debugger operations using the mouse. For more information on mouse capabilities, see the section "Using the Mouse in the Debugger".

The Debugger also provides some online help facilities. For more information on help facilities, see the section "Getting Help for Debugger Commands".

For complete information on the uses of these features and other Debugger features, see the section "Using the Debugger". For descriptions of all Debugger commands, see the section "Debugger Command Descriptions".

In general, you use the Debugger when:

- Your program triggers the Debugger because garbage — an unbound variable or too many arguments perhaps — was passed to a function, and you want to find out where the garbage came from. See the section "Analyze Frame Command".

- You want to see what's happening in the sequence of function calls just executed, including a history of these function calls, the argument values passed, the local-variable values, the source code, and the compiled code. See the section "Show Backtrace Command". See the section "Debugger Commands for Viewing a Stack Frame".

- You want to find out who or what is referencing a special variable or any other location in memory. See the section "Monitor Variable Command".

- You want to perform debugging operations using the mouse. See the section "Using the Mouse in the Debugger".

- You want to continue program execution, proceed from an error, restart a function, return from a function, or throw through a function. See the section "Debugger Commands to Continue Execution".

- Your condition handler does not work properly, and you want to debug this handler when it is encountered. See the section "Enable Condition Tracing Command".

- You want to edit your function's source code in Zmacs immediately after you have found the error. See the section "Edit Function Command".

- You want to put a Debugger backtrace into a mail message and send this message as a bug report. See the section "Mail Bug Report Command".

- You want to use Debugger breakpoint commands, instead of using the Trace facility or inserting a function in your code, to set Debugger breakpoints. See the section "Debugger Commands for Breakpoints and Single Stepping".

Overview of Debugger Commands

The Debugger offers more than 50 full-form English commands, which are implemented as CP commands. Debugger commands are entered inside the Debugger at the Debugger's command prompt, a right arrow (→). Commands fall into eight general categories:

- Commands for viewing a stack frame
- Commands for stack motion
- Commands for general information display
- Commands to continue execution
- Trap commands
- Commands for breakpoints and single stepping
- Commands for system transfer
- Miscellaneous commands

Most Debugger commands have corresponding key-binding accelerators, which means you can press a combination of one or more keys in place of the command. For example, you can press the accelerator `c-E` instead of entering the command :Edit Function.

Most Debugger commands also have keywords you can use to modify the command's behavior.

Many Debugger commands share the global command table. Therefore, you can enter these commands while you are in a CP command loop. You do not have to be in the Debugger. Note, however, that when you enter these commands while in the Debugger, you must type a preceding colon with every command; for example, you must type :Set Breakpoint in the Debugger.

These commands are:

- :Clear All Breakpoints
- :Clear Breakpoint
- :Disable Condition Tracing
- :Edit Function
- :Enable Condition Tracing
- :Monitor Variable
- :Set Breakpoint
- :Set Stack Size
- :Show Breakpoints
- :Show Compiled Code
- :Show Function Arguments
- :Show Monitored Locations

- :Show Source Code
- :Show Standard Value Warnings
- :Unmonitor Variable

For general information on using the Debugger, see the section "Using the Debugger". For documentation of each Debugger command, see the section "Debugger Command Descriptions".

Overview of Debugger Evaluation Environment

In the Debugger, you can evaluate a form as easily as you can in a Dynamic Lisp Listener read-eval-print loop. Evaluating a form in the Debugger, however, is particularly useful because you are evaluating the form in the context of a user program and the current stack frame. This means you can see the value of Lisp objects at the point in program execution where an error occurred or at the precise place in your program where you explicitly suspend execution and invoke the Debugger. You can even reference lexical (local) variables at the point where execution suspends.

Evaluating a form in the Debugger is a simple task. If you type a character other than the first character in a Debugger command — a colon or accelerator key — the Debugger immediately brings you into its evaluation environment. In other words, just type the form. Evaluation in the proper environment happens automatically.

For complete information on how to evaluate a form in the Debugger: See the section "Evaluating a Form in the Debugger".

Overview of Debugger Mouse Capabilities

When the output generated by Debugger commands is displayed in a Dynamic Window, it is mouse sensitive. You can perform several useful debugging operations simply by using the mouse to click on something. Some of these operations include: setting a breakpoint, monitoring a variable or another location in memory, evaluating a form, editing a function, setting the current frame, and choosing a proceed or restart option. The mouse documentation line at the bottom of the screen tells you what actions are available for the currently highlighted output item.

Besides performing certain mouse operations by clicking directly on displayed Debugger output, you can use menus to perform the usual large variety of other types of operations on Debugger output, just as you can with other kinds of output generated in the Genera software environment.

For more information on using the mouse in the Debugger: See the section "Using the Mouse in the Debugger".

Overview of Debugger Help Facilities

The Debugger provides online help for Debugger commands and their components, such as keywords. You can get help for all Debugger commands by typing c-HELP, which displays brief command descriptions and available key-binding accelerators. For more information about Debugger help: See the section "Getting Help for Debugger Commands".

Flavor Examiner

The Flavor Examiner enables you to examine flavors, methods, generic functions, and internal flavor functions defined in the Lisp environment. You can select the Flavor Examiner with SELECT X, or with the Select Activity Flavor Examiner command.

The Flavor Examiner lets you use all of the Show Flavor commands, saving the output in three history windows. Because much of the output is mouse-sensitive, it is convenient to use the mouse to select a flavor, method, or generic function from an output window to use as input to another Show Flavor command.

For a brief overview of the commands, see the section "Summary of Show Flavor Commands".

For a comprehensive description of each Show Flavor command, see the section "Show Flavor Commands".

Figure ! shows the initial window.

The Flavor Examiner window is divided into five panes.

Menu of Commands — the top-left pane

The top-left pane offers a menu of flavor-related commands, such as Flavor Components; this is the same as the Show Flavor Components command. You can choose one of these commands by clicking Left or Right. Clicking Left makes the command appear in the Command Input Pane. Clicking Right makes the command appear and also displays the command's arguments, in a form that you can edit.

The Help command displays documentation on the flavor-related commands. The HELP key provides information on all the CP commands you can enter.

The Flavor Examiner offers two commands for clearing and refreshing the display. The CLEAR DISPLAY command clears the display from the three output panes; it first asks for confirmation. The REFRESH DISPLAY command displays the information on the screen again.

When you click Left or Right on a command name, the command appears in the Command Input Pane.

Command Input Pane — the bottom-left pane

The bottom-left pane is a command processor window. If you click on commands in the Menu of Commands, the commands appear in this window. You can enter arguments (or commands) by typing them at the keyboard. This pane saves the history of all commands entered. You can click on the scroll bar to show different parts of the history.

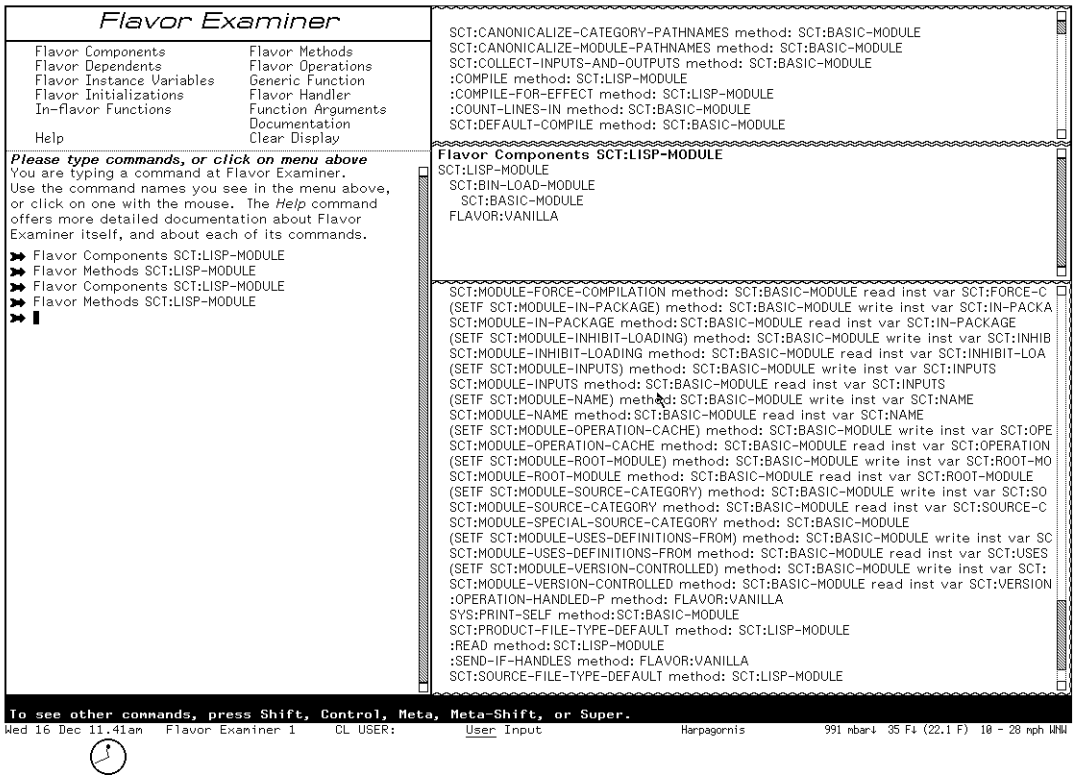


Figure 164. Flavor Examiner Window

You are not restricted to the commands in the Menu of Commands. You can give any command processor command.

The output of all commands appears in the Main Command Output Pane.

Main Command Output Pane — the bottom-right pane

Each command's output appears here. This pane saves the history of the output of all flavor-related commands. You can use the scroll bar to show different parts of the history.

Parts of the output of flavor-related commands are mouse-sensitive, so you can click on a flavor name or method name to enter it as an argument to another command.

If you give commands that are not flavor-related (such as the Show Host command), the output appears in a typeout window in the Main Command Output Pane. This kind of output is not saved in the history of this pane. The typeout window is itself a dynamic window with its own history.

When the output of the current command appears in the Main Command Output Pane, the output of the previous command is copied to the Previous Command Output Pane.

Previous Command Output Pane — the middle-right pane

This pane displays the output of the previous command. This pane does not save a history, but the second-to-last command is copied to the Second-to-Last Command Output Pane.

Second-to-last Command Output Pane — the top-right pane

This pane displays the output of the second-to-last command. This pane does not save a history. When another command is given, the contents of the Previous Command Output Pane are copied to this pane. Similarly, the contents of the Main Command Output Pane are copied to the Previous Command Output Pane.

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. 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 7 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)

Entering and Leaving the Inspector

You can enter the standalone Inspector via:

- Select Activity Inspector

- `SELECT I`

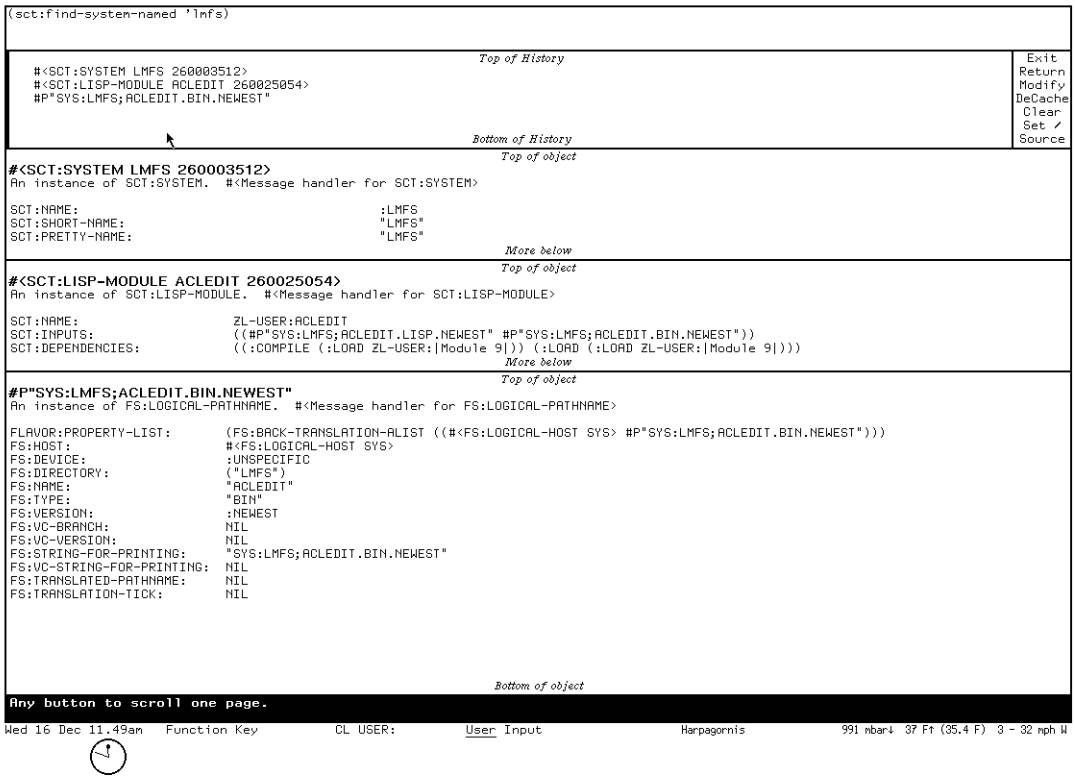


Figure 165. The Inspector

- [Inspect] in the System menu
- The Inspect command, which inspects its argument, if any
- 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.

See the section "The Inspector".

Document Examiner

Introduction to Document Examiner

Document Examiner is a utility for finding and reading documentation. The entire Symbolics document set can be read using Document Examiner.

You can look up documentation in Document Examiner's own window, in a Lisp Listener, or from the editor. No matter where you request to view documentation, Document Examiner manages all online lookup queries.

Using Document Examiner is similar to using the printed documentation. For example, just as books in the document set are available on a shelf or on your desk, Document Examiner, when first selected, displays the names of all the books in the document set. See Figure !.

You can open to a book's table of contents and find a topic to peruse. You can also open a book to the topic and read through to the end of that topic. Similarly, Document Examiner lets you scan a document's table of contents online and select the desired topic, or, if you know what you are looking for, you can "open" the documentation set directly to any topic and read to the end of that topic.

Just as you can browse through the index of a book to find the primary topic (and secondary topics) you want to read about, you can ask Document Examiner to show you all the topics that are indexed under a particular entry.

The big difference between the Document Examiner and a book or set of books is that Document Examiner provides powerful tools for moving from one kind of information to another.

Document Examiner allows quick, direct access to all levels of information:

- General, conceptual discussions of a topic
- Reference material on a topic
- Specific programming language information (for example, a Lisp function)

How Documentation Is Stored

The online documentation is kept in a *documentation database*, which consists of documentation binary files. When you boot your machine, the world load contains location information about the documentation topics, not the actual files themselves; thus, world size is increased by only a few hundred pages by having the document set available. The documentation database is stored on a file server; when you request to see a documentation topic, the appropriate part of the appropriate file is read in from the server.

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 record name. For example, the section you are currently reading is a record whose name is "How Documentation is Stored".

Records fall into two categories:

- *Object records* documenting Lisp code objects, such as **zl:setf** or **tv:menu**.

- *Concept records* documenting abstract ideas that are not tied directly to code, such as "Introduction to Document Examiner" or "Show Documentation Command".

Records have types. Examples of object record types are Lisp function, flavor, and variable. Concept records are called sections. The documentation lookup commands use the type indicator to distinguish between like-named records. For example, if you request to see the documentation for **error**, Document Examiner lets you know that documentation exists for **error** as both a flavor and as a function.

Records have *topics*. The Show Documentation command looks up records by their topics. The topic uniquely identifies a record.

Records also have *keywords* with them. A keyword is comparable to a word in an index entry. This record contains the keywords "documentation record". The Show Candidates command compares your lookup request to all the keywords and documentation topic titles in the database. So, for example, if you want to find out what information is available on the general topic "record", Document Examiner includes the record you are now reading in its list of possible choices because "record" is one of its keywords. You can see the keywords for any record in the overview of the record.

Help in Document Examiner

Press the HELP key in the Document Examiner window to display a summary of commands. Clicking Left on [Help] in the command pane produces the same abbreviated display.

Complete documentation of all features is available by command is equivalent to clicking Middle on [Help] in the command pane or by typing Document Examiner Documentation at the prompt.

Document Examiner Window

You can select Document Examiner in one of two ways:

- Keyboard: Press SELECT D or type Select Activity Document Examiner at the Command Processor.

- Menu: Click on [Document Examiner] in the System menu.

The Document Examiner window contains the following panes, as displayed in Figure 166:

- *Viewer* — where the documentation is displayed

- *Current candidates* — list of current choices, usually resulting from a Show Candidates or Show Table of Contents command

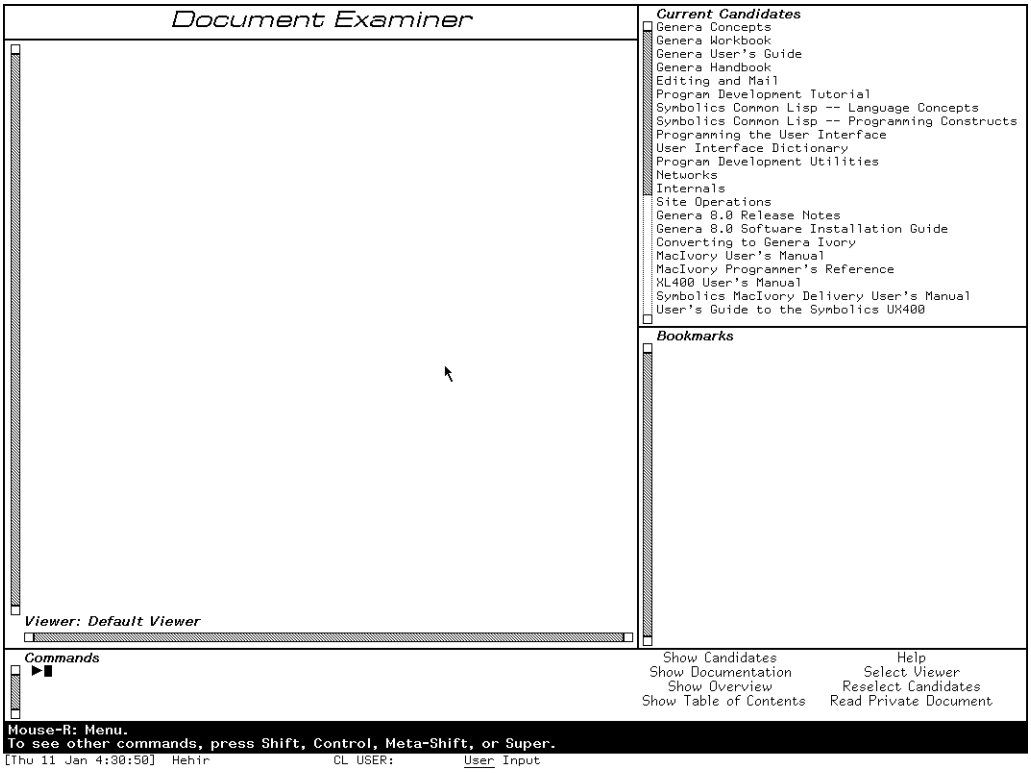


Figure 166. Upon selection, the Document Examiner displays all the books in the document set.

- *Bookmarks* — list of topics you've looked at or indicated that you wanted to look at

- *Command* — where you enter commands, either by mouse or by typing a command

Each of these panes is discussed in detail in this section.

Viewer Pane

The large area on the left of the window is called the *Viewer pane*. All the documentation you select to read is displayed in the viewer.

When you select a topic for viewing in Document Examiner, the topic is displayed at the end of the viewer and the topic's name is added to the list of bookmarks. Topics displayed in a viewer are separated by a light horizontal line.

Certain items displayed in the viewer are mouse sensitive; for example, clicking Left on cross-references or the names of documented Lisp objects displays their documentation in the viewer, clicking Middle displays an overview of the topic (a

summary of where it is and what it is), and clicking Right presents a menu of operations. Check the mouse documentation line for details.

The viewer is a dynamic window, which gives you the ability to copy a region of text from the viewer into another window, such as Zmacs or Zmail. You can mark a region with the mouse by pressing the CONTROL key and push the marked text on the kill ring using s-W. For example, you can extract an interesting Lisp code example from a documentation topic and run it in an editor buffer or Lisp listener. Or you can include selections from the documentation in mail or text files. You can also search for a particular string using s-R to to search backward and s-S to search forward. For an introduction to using dynamic windows, see the section "Using Your Output History".

When you first select Document Examiner, you see the *default viewer*. It is labeled as such in the lower left-hand corner of the pane. You can create or select another viewer, just as you can create or select another editor buffer. To create a new viewer click Left on [Select Viewer] or enter the Select Viewer command and name a nonexistent viewer. You can see a mouse-sensitive list of existing viewers by pressing c-?. To go between two viewers, press c-m-L. To select another viewer, use Select Viewer.

When you look up documentation in a Lisp Listener (using the Show Documentation command) or in the editor (using m-Sh-D or m-X Show Documentation), bookmarks for those topics are inserted in a special viewer called the *Background viewer*.

Examples of Lisp code whose lines are wider than the viewer display with those lines wrapped around. When you need to see such examples in a more readable layout, use the Show Documentation command in a wider window, such as a Lisp Listener.

Current Candidates Pane

The upper right-hand pane of Document Examiner's window displays the names of those topics that are the *current candidates*, that is, a menu of choices from which you can select a topic for viewing. These candidate lists are produced by the Show Candidates and Show Table of Contents commands.

When you first enter Document Examiner, the current candidates pane displays the list of documents registered in the documentation database, as displayed in Figure 166. Each time you use Show Candidates or Show Table of Contents, a new list of current candidates is generated and displayed. Note that lines that are wider than the current candidates pane are truncated.

You can redisplay a list of candidates not currently showing by clicking on [Reselect Candidates] or using the Reselect Candidates command. You'll see a menu of all the documentation searches conducted during the current session. This list is helpful when, for instance, you need to cycle through several sets of candidates. Instead of having to remember exactly what you asked for each time you want to look at it, use the Reselect Candidates command and click on the candidates pane that you want to see.

Bookmarks Pane

The lower right-hand pane of the Document Examiner window contains a list of *bookmarks*. Bookmarks are the names of topics that you have displayed in the viewer or the names of topics that you want to display later on. Thus, a Document Examiner bookmark is similar to a bookmark you might place in a printed book — as a reminder of something you have just read and might want to reread or as a pointer to something you should read in the future.

Document Examiner automatically inserts a bookmark for a topic when that topic is first displayed in the viewer. The list represents a history of your selections for a particular viewer. You can also explicitly add a bookmark to the list of bookmarks without having the topic displayed in the viewer. See the section "Putting Topics Aside for Future Reading".

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. See Figure !.

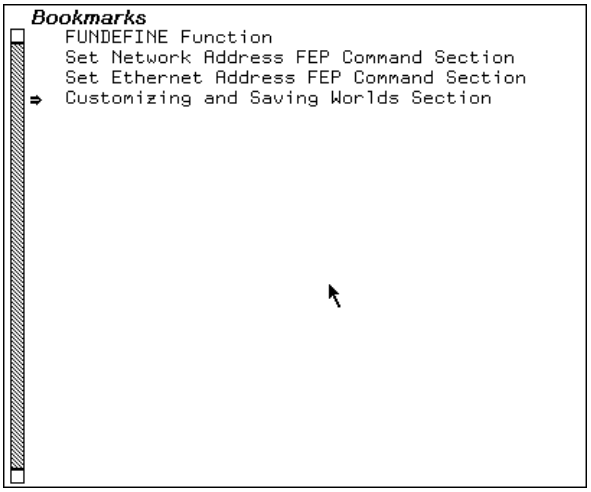


Figure 167. The bookmarks pane shows which topics have been displayed in the viewer.

Names that are wider than the bookmarks pane are truncated.

You also use a set of bookmarks in creating a private document. See the section "Creating a Private Document".

Command Pane

The bottom portion of the Document Examiner window is called the command pane. It accepts hand-typed commands at the triangular Command Processor

prompt to the left and displays a menu of Document Examiner command menu items to the right.

The Command Processor offers completion on command names as well as topic names. The HELP key (after you type at least one character) displays a mouse-sensitive list of possible completions. Pressing HELP before you start to type a command name displays the available commands. After you type at least one character, c-? displays a mouse-sensitive list of commands *starting with* the character or characters you have typed and c-/ displays a mouse-sensitive list of commands *including* the character or characters you have typed.

The mouse documentation line, situated below the command pane, tells you how to use the mouse to activate a particular command from the menu.

The following list describes the commands. All these commands are also available as menu items.

- Help — offers a brief command summary. Or click Right on [Help].

- Document Examiner Documentation — displays the full Document Examiner documentation. Or click Middle on [Help].

- Show Candidates — searches the entire database for documentation topics with titles or keywords that resemble or match what you're looking for and displays a list for you to choose from.

- Show Documentation — displays the documentation for a specific topic and is useful when you know exactly what you're looking for.

- Show Overview — displays basic information about a topic, including a tree graph that shows how the topic fits in with other topics and what keywords are specified for it.

- Show Table of Contents — displays the table of contents of a topic, that is, what other topics that topic includes. It is most useful for getting information about an entire chapter, section, or book.

- Reselect Candidates — displays a menu of all the candidates lists created for a particular viewer during the current work session. Candidates lists are created whenever you invoke the Show Candidates or Show Table of Contents commands. The display is a mouse-sensitive menu from which you can redisplay the results of a previously executed command.

- Select Viewer — changes from one viewer to another or creates a new one. You can create a new viewer by naming a nonexistent viewer in the Select Viewer command. You can move between two viewers by pressing c-m-L.

- Remove Viewer — removes a viewer from the list of viewers. Also available from the menu presented when you click Right on [Select Viewer].

- Hardcopy Viewer — hardcopies the contents of a viewer. Also available from the menu presented when you click Right on [Select Viewer].

- Save Private Document — saves the list of bookmarks in the current viewer as a private document. This means you can create your own private document by creating bookmarks for one or more documentation records and then issuing this command to write the collection of bookmarks to a file. Also available from the menu presented when you click Right on [Read Private Document]. You can read or load a private document when you need it. It is not part of the documentation database. See the section "Creating a Private Document". Read Private Document — allows you to view private document collections created from the documentation database. This command loads all the text associated with the private document into a viewer.

- Load Private Document — loads the bookmarks associated with a private document, but does not load the text. Also available from the menu presented when you click Right on [Read Private Document].

- Hardcopy Private Document — sends all the text in a private document to a printer. Also available from the menu presented when you click Right on [Read Private Document].

Looking Up Documentation

When you look through a printed book for information, you probably scan the table of contents to see how the book is organized and maybe skim a few chapters to see their major sections, perhaps inserting a bookmark or two when things look interesting. You might then look over the index to see the number and kinds of main and related entries on a particular subject. Sometimes you know exactly what you are looking for and can go directly to the section you want to read.

Document Examiner provides a variety of commands and tools for online lookup that are analogous to the methods you use to examine a printed book, but are much more flexible. They are not limited to a single book, however, and they allow you to browse in the entire documentation database with just a few keystrokes and to take advantage of some "smart" aids to readers, such as the Show Candidates command.

Requests for online lookup are always made with respect to a *documentation topic name*, which can be anything from the name of a document, chapter, or other book subdivision, to the name of a Lisp code object, like a function or flavor. Every Lisp object is documented in its own documentation record; the documentation topic name of the record is the name of the Lisp object. (You may need a package prefix. See the documentation for languages other than Lisp in case there are special procedures.)

Show Documentation
: Displays the documentation for a specific topic name in the viewer and inserts a bookmark for the topic. You can do the same thing by clicking Left on the name of any documentation topic. You could look up the section you are reading now by using Show Documentation and specifying "Looking Up Documentation".

Show Candidates
: Displays, after searching the database, all topic names that satisfy your lookup request. This is often the best way to search the documentation database. By default, Show Candidates uses a heuristic, or "smart", searching strategy that allows you to see all the candidates that are close to what you asked for. For instance, if you specify "local" to Show Candidates, the candidates list includes "locals" "locative", "location", and "localize", as well as "local". Show Candidates lets you specify several search strategies. For information on these alternative search strategies, see the section "Show Candidates Command".

Show Overview
: Displays the local context of a topic name, showing which topic includes it in the document hierarchy, which topics are on the same level, and which topics are included by it in the hierarchy. This is good for examining the organization of topics and tracing topics to more general and more specific levels of abstraction. The display also provides related information, such as, which documents a topic appears in and what keywords are associated with it.

Show Table of Contents
: Displays the table of contents for a given topic name, showing the supplied topic name and all of its subordinate topics. This command is good for examining the organization of topics within a document.

Genera's online documentation facility is flexible. You can look up information from several contexts — in a Lisp Listener, in an editor buffer, or directly in Document Examiner. Where you look up documentation probably depends on the kind of work you are doing and your own particular workstyle.

- If you are programming in an editor buffer, you can use m-X Show Documentation or its keyboard accelerator, m-sh-D.

- In a Lisp Listener, you can issue Show Documentation at the Command Processor, m-sh-A to see documentation for a Lisp function, m-sh-F to see documentation for a flavor, or m-sh-V to see documentation for a Lisp variable.

- If you plan to poke around or to read a great deal of documentation, you can take advantage of the extra features offered by Document Examiner, such as bookmarks and the Show Candidates command. Shifting to Document Examiner

in such cases is easy. Bookmarks for all the topics you've looked at outside Document Examiner are inserted in the Background viewer.

In addition, Document Examiner allows you to:

- Insert a bookmark in a documentation topic, see the section "Putting Topics Aside for Future Reading".
- Display the documentation for topics that are explicitly cross-referenced, see the section "Looking Up a Specific Topic".
- Display the documentation for implicit cross-references, such as the following mouse-sensitive reference to **zl:string-capitalize-words**.

Any topic name appearing in any pane of the Document Examiner window is *mouse-sensitive*. That is, when you move the mouse over a topic name, the mouse highlights the name in an outline box; clicking one of the mouse buttons on a highlighed topic name causes some action to occur. Clicking Left displays the documentation and clicking Middle displays the overview.

Clicking Right brings up a menu of operations that can be performed on that topic name:

View the documentation
Hardcopy documentation
Display a table of contents
Insert a bookmark
Display an overview
Discard a topic

The lookup commands distinguish among like-named topics. For example, if you use Show Documentation for **evalhook**, Document Examiner asks you to select **evalhook** as a function or as a variable. See Figure !. Similarly, the lookup commands distinguish between Zetalisp objects and those that are Common Lisp or Symbolics Common Lisp objects; Zetalisp objects are displayed with the **zl:** package prefix.

Looking Up a Specific Topic

Document Examiner provides several ways to look up documentation for a specific topic, once you know the right topic name:

- Use the Show Documentation command.
- Use the mouse to display the documentation for cross-references and the names of Lisp objects. All topic names are mouse-sensitive; clicking Left on a topic name displays the associated documentation. The names of Lisp objects mentioned in the documentation appear in a boldface character style.

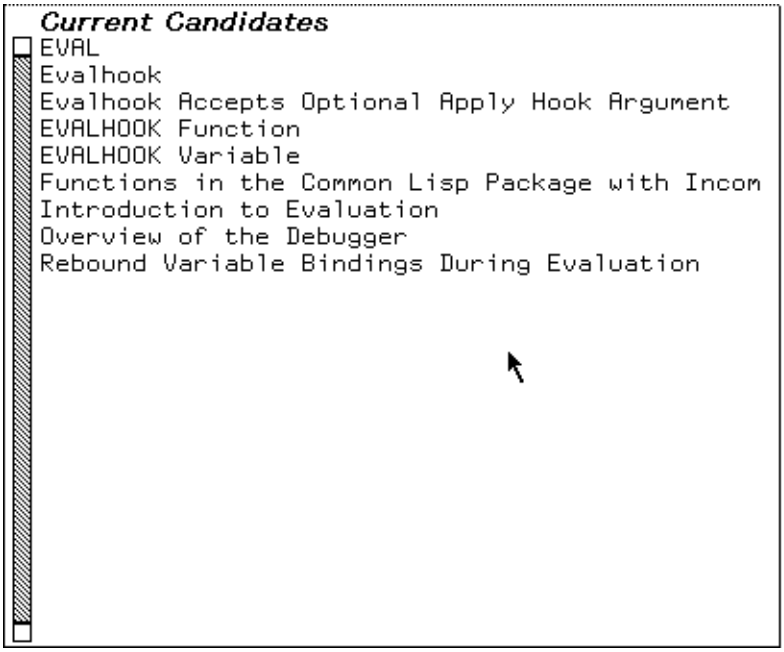


Figure 168. Document Examiner lets you choose among like-named topics.

Example: the topic ***load-set-default-pathname***, displayed in the viewer, mentions the variable ***load-pathname-defaults***, as shown in Figure !. Clicking Left on the variable name displays the documentation for the topic "*load-pathname-defaults*". See Figure !.

- Use the mouse to display the documentation associated with current candidates, bookmarks, or topic names displayed in an overview or a table of contents.

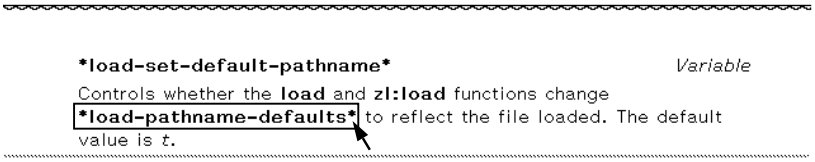


Figure 169. The names of Lisp objects displayed in the viewer are mouse-sensitive.

In addition, you can look at documentation topics in the editor and in a Lisp Listener.

Show Documentation Document Examiner Command

Show Documentation is useful when you know exactly what you are looking for. Show Documentation looks up a topic, displays it in the viewer, and, if it is a new

topic, inserts a bookmark for the topic in the bookmarks pane. To invoke the command you can either click on [Show Documentation] in the command menu or type Show Documentation in the command pane. The command prompts for a topic name.

Clicking Left on a topic is the equivalent of Show Documentation.

If you aren't quite sure what you're looking for, see the section "Show Candidates Command".

Topic names for methods are of the form (flavor:method *:generic-function-name flavor-name*). Thus, to use Show Documentation for a method, give the topic name of the method in this form, for example:

```
Show Documentation (flavor:method :set-edges tv:menu)
```

Show Documentation is also an editor command and a Command Processor command. In an editor, m-X Show Documentation (or its keyboard equivalent m-sh-D) prompts you for a topic name, with completion.

When you are typing in a Lisp Listener or a Lisp mode buffer in Zmacs, the following commands give you documentation for the current Lisp object (the one nearest the cursor):

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.

You can use c-/ and c-? to find topic names containing or starting with the the sequence of characters you have typed so far. For example,

Show Documentation search c-/

displays a mouse-sensitive list of all documentation topics *containing* "search" in their names. And,

Show Documentation search c-?

displays a mouse-sensitive list of all documentation topics whose names *start with* "search".

See the section "c-? and c-/".

Searching Through Documentation

In addition to looking up specific topics online, you can search through the entire documentation database. Document Examiner includes many tools to make this searching easy.

The Show Candidates command, for instance, can search for a broad range of possible documentation topics using heuristic, or "smart", techniques. This means that the command locates a word and variations on it instead of searching for a strict

match. The command defines the different variations for the word by checking for common suffixes using a technique called *stemming*. Thus, if you specify "local" to Show Candidates, the result includes "locals", "locative", "location", and "localize", as well as "local".

Show Candidates isn't limited to documentation titles in doing the search. It searches through an index that contains all the words in topic titles and a list of *keywords* (somewhat like an index entry) for each documentation topic.

If, in the course of your searching, you see topics that look interesting, you can either read them as you go or add bookmarks so you can look at them later.

Document Examiner also helps you search by enabling you to jump from topic to topic.

- You can click on implicit cross-references to language objects, such as **setq**, **cdaddr** and **zl:string-pluralize**.

- You can click on explicit cross-references, such as, see the section "String Conversion".

Searching with the Show Candidates Command

The Show Candidates command is your first choice for browsing through the documentation database. While Show Candidates is fundamentally an index-search command, it has several capacities beyond what is normally found in index-searching.

- In the first place, Show Candidates searches through all the topic names in the database, and also searches through all the keywords (which are similar to index entries) as well.

- Secondly, by default, Show Candidates performs "smart" (heuristic) searching. This means that you no longer have to think about the form of the word to supply as a lookup request; the heuristic approach finds singulars and plurals, gerunds, negations, and so on. This is because the search is based on the *stem* of your lookup request.

  For example: The stem of "move" is "mov". Show Candidates `move` will match any topic names and keywords containing "move", "moves", "moved", "moving", *move-mumble*, ":move", and so on. (This is a hypothetical example. In fact *move-mumble* is not documented.)

If you supply two or more words to Show Candidates, it searches for candidates that match *all* stems in any order. Thus, if you supply both "pathname" and "completion" to Show Candidates, your candidates list includes both **fs:complete-pathname** and "Pathname Completion is Supported". This is the default behavior.

You can specify adjacent search and also string searching by exact match, initial string, and substring by typing in your search term or terms and then pressing `m-COMPLETE` to display a menu from which you can choose those options. Press `END` to accept the choices displayed in boldface or `ABORT` to abort.

If you select adjacent search and supply "pathname" and "completion" in that order, your candidates list will include only "Pathname Completion is Supported".

Use the same menu to limit your search to exact matching, initial substring matching, and substring matching.

Putting Topics Aside for Future Reading

Occasionally in browsing through the database you notice topic names that look interesting but that you cannot or do not want to read immediately. Suppose, for example, that you are scrolling through several hundred current candidates generated by the Show Candidates command. Rather than read the documentation for each interesting topic as you come across it, you prefer to scan all the candidates first, mark all those topics that look like they might be helpful, and read them all at once later. Remembering their topic names would be extremely tedious.

Document Examiner lets you put documentation aside for future reading by allowing you to explicitly insert a *bookmark*, or pointer, for a documentation topic. The topic name is added to the bookmarks pane. To insert a bookmark click sh-Middle on a mouse-sensitive topic name.

Bookmarks are automatically inserted in two cases: whenever you read the documentation for a topic into a viewer and when you look up documentation in a Lisp Listener or in the editor. In the latter case, the bookmarked is inserted in a special viewer called the *Background viewer*, but the topic is not displayed there.

Bookmarks are used to create your own document for online viewing, see the section "Creating a Private Document".

Particular bookmarks are associated with a particular viewer, so that when you select another viewer, the list of bookmarks associated with it is also selected.

Getting Information About a Documentation Topic

Two commands — Show Overview and Show Table of Contents — are useful for examining the organization of topics within a document and for learning the position of a particular topic in the book or books in which it appears.

Show Overview

Sometimes you notice an intriguing-looking topic name among a list of current candidates but are not quite sure that the topic is the one you need. Before you commit yourself to reading it into the viewer, you might want to see how the topic fits into the structure of the book or books in which it appears or which other topics, related to it, are covered in the same book. Other times you might read a topic into the viewer and find that it is not quite at the right level of abstraction: it provides too much reference information, whereas you need more general coverage.

Document Examiner provides a command — Show Overview — that displays the position of a topic within the document or documents in which it appears. Given a topic, the command displays a two-part overview of the topic in a dynamic typeout window:

The top part includes:

- The type (is it a section of text or does it document a Lisp object, like a function, for instance) and the 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
- The topic's keywords (similar to index entries)
- The names of related topics (cross-references)

The names of topics and books in this display are mouse-sensitive.

The bottom part is a graph of the document hierarchy around the topic you choose. (If the topic appears in more than than one document — multiple graphs are displayed.) This graph includes:

- The topic that includes the original topic
- Other topics included by the topic that includes the original topic
- The topics the original topic includes

Figure !shows the display produced by doing Show Overview of the topic "Disk Error Handling".
All the items in the overview graph are mouse-sensitive.

For example, when you get an overview of "Disk Error Handling", you can see that the topic is included in the topic "Disk System User Interface" and that it includes four other topics. If this overview does not give you enough information, try an overview on the parent topic (in this case, "Disk System User Interface").

While the Show Candidates command searches through the entire documentation database, Show Overview helps you find out where you are in a document and how to navigate your way through it.

Show Overview differs significantly from "Show Table Of Contents" . Compare the display of the two commands. See Figure !.

You can also find the printed book in which the topic appears by using the m-X What Document command in the editor. It 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.

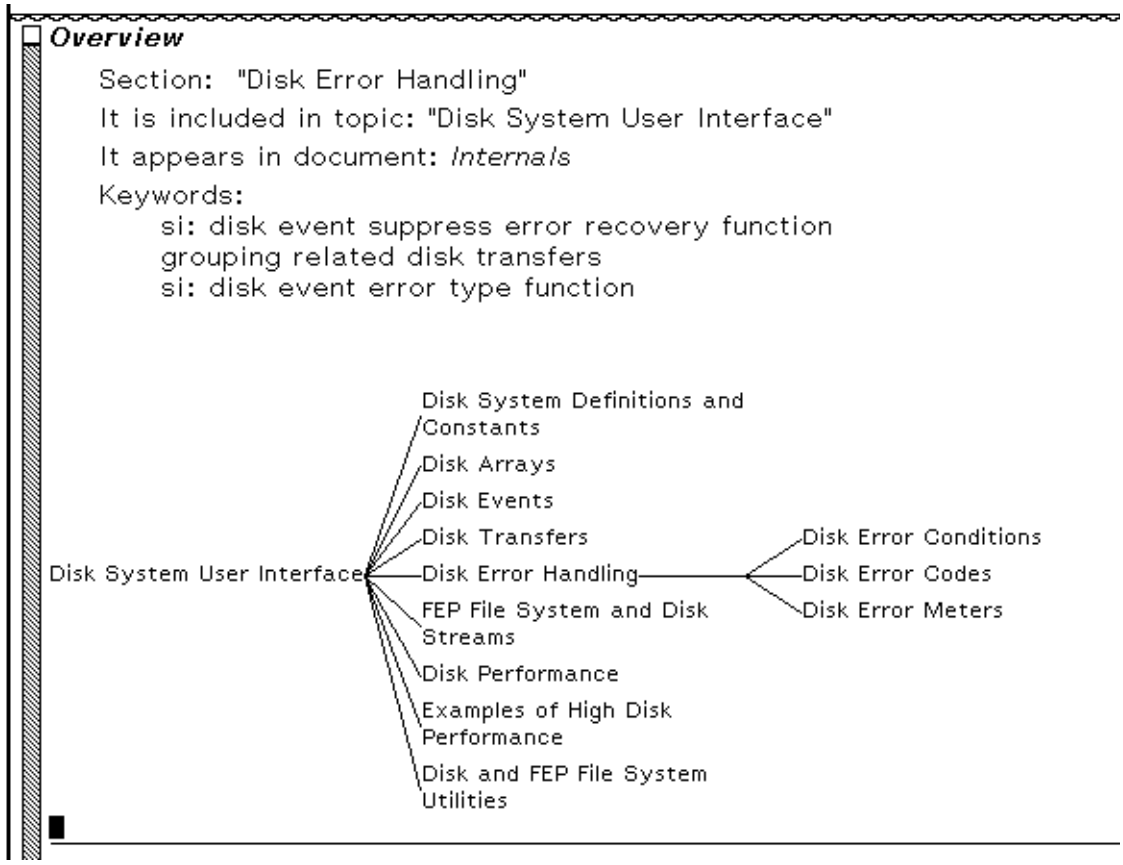


Figure 170. Document Examiner display of Show Overview of topic "Disk Error Handling".

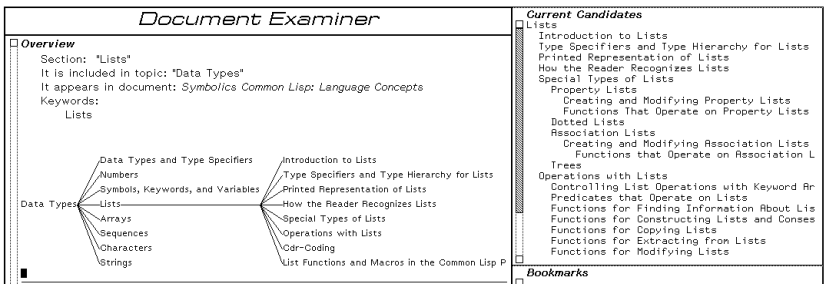


Figure 171. The candidates pane shows the table of contents for a topic; the type-out window shows the overview of the same topic.

Show Table Of Contents

Much as with printed documentation, you might want to become familiar with the overall structure of an online document before reading about any one topic. When you examine documentation online, the more general the topic you look up, the

larger the amount of documentation you see for it. The most general topic names, obviously, are the names of the books in the printed documentation set. For example, if you asked to see the documentation for the topic name "Genera User's Guide", Document Examiner would display the contents of the entire book. This might not be what you want!

The Show Table of Contents command is useful on those occasions:

- When you are unsure what level in the documentation you need (chapter level or section level, for example)
- When you would like to become familiar with the organization of a document

Given a documentation topic name, the command displays the topic name and all the topic names subordinate to it in the candidates pane. For example, Figure ! shows the first pane of the table of contents of the document "Genera User's Guide".

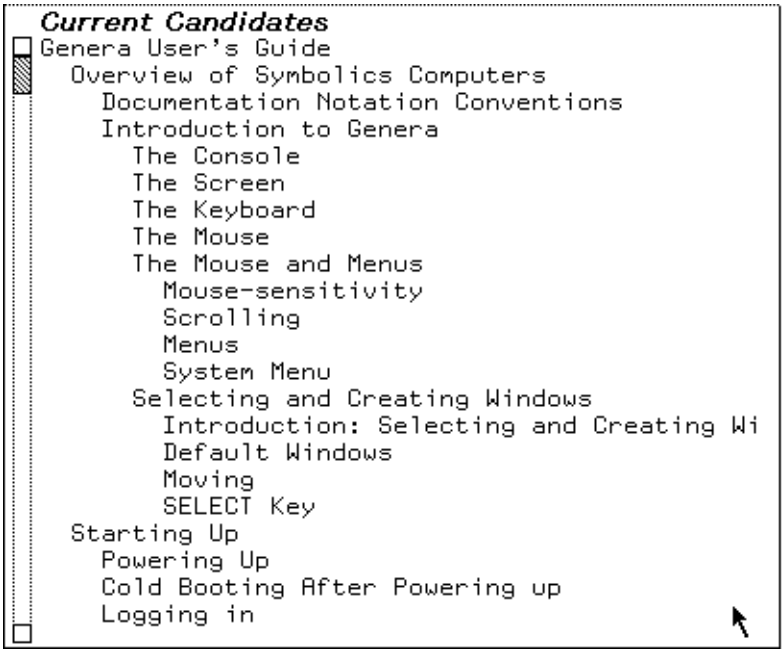


Figure 172. The candidates pane displays the table of contents for an entire document.

The indentation of the candidates indicates the hierarchy of topics.

You can ask to see a table of contents for any topic, not just a book. Of course, if a topic doesn't contain any other topics, then it has no table of contents.

This command is also available in the editor as m-X Show Table of Contents. c-U m-X Show Table of Contents prints the table of contents on a printer.

For more general searching, see the section "Searching with the Show Candidates Command".

Hardcopying Documentation

Whenever you are looking up documentation using Document Examiner, you have the option of hardcopying that documentation.

There is a Hardcopy Documentation command available from the Document Examiner command pane. The same command is also available from the menu displayed when you click Right on a topic name.

At a Command Processor, Show Documentation prompts you for a topic name, with completion. When you include the keyword argument `:Output Destination`, the command offers to route the documentation to the default text printer.

In the editor, you can hardcopy a topic by issuing Show Documentation with a numeric argument, for example:

```
c-U m-X Show Documentation catch-error-restart
```

Organizing Topics in Document Examiner

You do not need to put all the documentation you ever looked at or wanted to look at in a single Document Examiner viewer. You can create multiple viewers to help you organize your use of Document Examiner. You can also create private documents that you can load into a viewer at any time, even if you have rebooted since you last looked at Document Examiner.

Using Multiple Viewers

When you select a topic for viewing, the topic is displayed at the end of the current viewer, separated from the previous topic by a horizontal line. After a day or two of browsing and reading documentation, the viewer becomes cluttered with dozens of topics. The list of bookmarks gets long enough that you have to to scroll through it to locate a topic name.

As an alternative to this hodgepodge of numerous, unrelated topics in one viewer, you might find it more convenient to organize the results of browsing into different viewers. For example, make random lookup requests in the default viewer (the one that appears when first selecting Document Examiner), but read all documentation on a particular subject, like "Arrays", into its own viewer, called Array Documentation.

You can create as many viewers as you want with the Select Viewer command, also available by clicking Left on [Select Viewer] in the commands pane. Whenever you create a new viewer, the default list of current candidates is all the books registered in the documentation database. Note that each viewer has its own list of bookmarks; bookmarks in another viewer do not transfer to the new viewer. The Reselect Candidates command also applies to the current viewer only.

Use Select Viewer also to return to an existing viewer. If you cannot recall its name, use c-? to display a menu of viewer names. Use c-m-L to move between two viewers.

This system of organizing lookup requests makes it easy, for instance, to hardcopy all the documentation on a particular subject: just invoke the Hardcopy Viewer command or click Right on [Select Viewer] and then click on [Hardcopy Viewer] from the menu. Otherwise you would have to individually locate every topic and hardcopy it, a rather tedious process. In this scheme discarding collected topics you no longer need is just as easy: use Remove Viewer or click Right on [Select Viewer] and then click on [Remove Viewer] on the menu to delete the entire contents of a viewer.

Creating a Private Document

You might decide that you want to save the documentation in a viewer beyond the current work session (the span of time between cold-booting), so that the next time you invoke Document Examiner you will not have to painstakingly repeat the lookup process.

Document Examiner provides a mechanism for saving a viewer's documentation topics in a file called a *private document*. A private document is not part of the central documentation database. A private document is accessible on demand for online viewing simply by reading the file into Document Examiner.

Strictly speaking, a private document is a collection of bookmarks, rather than a collection of the actual documentation topics. Thus, to create a private document you must first create a list of bookmarks and then save it in a file. For example, to create a private document on login procedures and functions, use the following procedure:

1. Create a list of bookmarks on the appropriate topics, like

   ```
   login-forms
   login-setq
   System Initialization Lists
   zl:login
   ```

   by looking up some topics or by clicking appropriately on the list of candidates or on mouse-sensitive items in the viewer.

2. Discard extraneous topic names, if any, from the current viewer, since Save Private Document writes *all* the bookmarks in the pane to the file. Click Sh-Middle on a bookmark to remove the topic. Alternatively, you can insert the bookmarks for your document in a new viewer.

3. Save the bookmarks in a file by invoking Save Private Document or clicking Right on [Read Private Document] and selecting [Save Private Document] from that menu. When prompted, supply the pathname of a file to contain the private document's bookmarks. The following is the prompt for Save Private

Document:

```
(The pathname of a file [default
PURPLE:>prince>private.psb.newest]):
```

The default location for a private document is always your home directory. Thus, with a home directory of PURPLE:>prince>, if you give Save Private Document the filename "login-book", the command writes the list of bookmarks to PURPLE:>prince>login-book.psb.

You can read, load, or hardcopy a private document at any time. Reading a private document into Document Examiner means that the documentation is loaded into your computer and displayed in the viewer of your choice. Use Read Private Document or click Left on [Read Private Document]. You can also load the bookmarks for the private document into the Document Examiner without having the documentation displayed. Then all you have to do is click as usual to display it. To do this, use Load Private Document or click Right on [Read Private Document] and select [Load Private Document] from the menu.

You can also print a copy of a private document by clicking Right on [Read Private Document] and selecting [Hardcopy Private Document] from the menu.